UNIVERSITÀ DEGLI STUDI DI FIRENZE
Facoltà di Ingegneria - Dipartimento di Sistemi e Informatica

---

Dottorato di Ricerca in Ingegneria Informatica, Multimedialità e Telecomunicazioni
ING-INF/05

Formal Methods and Code Generation Techniques in the Development of Railway Signalling Systems

Candidate

Gianluca Magnani

Supervisor

Prof. Alessandro Fantechi

Ph.D. Coordinator

Prof. Luigi Chisci

Ciclo XXV, 2010-2012

Abstract

This dissertation is the result of three years of experimentation and research at General Electric Transportation Systems (GETS), Intelligent Control Systems, division of Florence. In the early years of 2000 GETS decided to open to research activities in order to introduce formal methods in the development cycle of railway signalling solutions. For this purpose, it was decided to start a collaboration with the University of Florence, in particular with the Computer Engineering Department (D.S.I. - Dipartimento di Sistemi e Informatica). This work reports the achievements of the research, the lesson learnt and, finally, the advantages of the introduction of formal methods in GETS development cycle.

Contents

List of Figures

CHAPTER 1

Introduction

The industries are increasingly focusing on the adoption of formal and semi-formal methods, including modeling technology, in the different stages of the development of software products [7] [48] [47]. The Paris Metro [23], the SACEM system [35], and the San Juan metro [43] are past and recent examples of successful stories about the usage of these technologies in the railway domain. The use of models allows to reach an abstraction level difficult to achieve when working with hand-written code, and allows to grasp concepts that otherwise would be lost in the early stages of development. The many benefits of this approach become focal for the development of safety-critical systems, where having control of the system that is being developed is critical.

The industries involved in the development of safety related systems, such as railway, aerospace, automotive, moved closer to graphical tools such as ASCET, AutoFocus [38], SCADE [19], and Simulink/Stateflow [82]. In particular, Simulink/Stateflow is widely used for the development of embedded control applications, thanks to the built-in tools Stateflow Coder [81] and Real Time Workshop Embdedded Coder [80] which allow the generation of code starting from models.

General Electric Transportation Systems (GETS) is a well known railway signalling systems manufacturer leading in railway signalling systems, from Automatic Train Protection systems (ATP) to Interlocking Systems (IXL). In early 2000's GETS decided to introduce formal methods in its development process, initially by the means of Simulink/Stateflow for modeling requirements, then for code generation starting from the model, and finally applied to higher levels of design, such as requirement and architecture specification.
GETS products are developed mostly for Europe and shall hence comply with the CENELEC standards [9] [10], a set of norms and methods to be used while implementing a product having a determined safety-critical nature. In order to certify a product according to CENELEC, companies are required to give evidence to the certification authorities that a development process coherent with the prescriptions of the norm has been followed. The introduction of formal methods in the development process is thus not easy: for example, in the case of automatically generated code from formal descriptions, the code shall anyway conform to specific quality standards, and normally the companies use coding guidelines in order to avoid usage of improper constructs that might be harmful from the safety point of view. Thus, when the Model-Based Development with code generation is adopted, the generated code shall comply to the same standards asked for the hand-crafted code. Furthermore, the problem is raised also for the used tools used code generation tools, since the norms ask for a certified or proven-in-use translator from models to code: Simulink/Stateflow do not possess this property, and thus a strategy has to be defined in order to address this issue.

Once the Model-Driven Development was accepted formally in the development process of the company, the research activity moved its focus into the introduction of modeling techniques and formal methods also in early phases of the development process, such as requirements and architecture specification, in order to create an uniform development process.
The need for a more formal approach to high-level design derives, among the others, from the inadequacy of a text centric approach, which usage has been reduced in favor of a more formal notation, traceability enforcement and cross-

phase integrated documentation obtained through the usage of SysML/UML graphical languages (Object Management Group [OMG], [53]).

The dissertation, which reports the results achieved in the context of the collaboration between GETS and University of Florence, is structured as follows. Chapter 1 presents an overview of the normative background in which GETS operates, an introduction to the definition of formal methods and the objectives of the research and collaboration with GETS. In Chapter 2 is given a short introduction to the Model-Based Design and to the Simulink/Stateflow toolsuit. In Chapter 3 it is presented the Model-Based Design approach applied to the GETS development process of software products, result of the research and the collaboration with University of Florence, and a case study in the context of the SSC Baseline 3 GETS developed project, is detailed in order to better understand the technologies applied to the new development process. In Chapter 4 it is presented the second direction of the research, the introduction of formal methods in the early phased of the development, i.e., the requirements and architecture specification through the usage of SysML language. Chapter 5 summarizes the leasson learned during research and the experience in collaboration with GETS.

1.1 Background

1.1.1 Formal methods

With the term *formal methods* we refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems [15] [5] [49]. Mathematically rigorous means that:

- specifications used in formal methods are well-formed statements in a mathematical logic;
- the verifications are rigorous deductions in that logic;

The formal methods applied to development process usually result in a framework that includes a formal language, verification techiniques and a set of tools that implements all the above. The formal language is an unambiguous notation which possess well-defined semantics and syntax, used to express a system

specification. The verification techniques are applied basing on the system specification in order to check if the system conforms to the desired properties. Finally, the tools are intended as a development process which is composed by successive refinements in which the intial specification is transformed into a more detailed representation of the system until the final implementation, included the verification of the system properties.

Formal methods have been widely discussed in literature, such as the VDM (Vienna Development Method) [41], B method [69], and the RAISE (Rigorous Approach to Industrial Software Engineering) [51] method, together with formal methods that focus mostly on parts of the whole formal process, such as PVS (Prototype Verification System) [58], HOL (Higher Order Logic) [32] and Z [73]. Unfortunately these methods did not widely spread in industrial employment, since, except for few projects such as Paris Metro onboard equipment [23], where the B method has been employed, and the Maeslant Kering storm surge barrier control system [86], where both the Z and the Promela [37] notations have been used, there is scarce in industrial context. Industries perceive formal methods as experimental techologies, and scepticism about their usefulness remains widespread.

1.1.2 Norms in the design and development of safety-critical railway systems

The Safety Critical systems are those systems whose exercise involves a level of risk of exposure of people, environment and material assets to dangerous situations, with the possibility of accidents due to malfunctions caused by errors or failures [9] [11] [10]. It is necessary to be sure that a system has a probability less than a given tolerable limit (THR, *Tolerable Hazard Rate*) to cause serious injury, before it can be put into operation. *Safety* is therefore defined as the *absence of unacceptable levels of risk* [9], or even the property of a system to not cause harm to human life or to the environment [70].

In the field of railway signalling, standards were created by CENELEC (*Comité*

européen de normalization en électronique et en électrotechnique or *European Committee for Electrotechnical Standardization*), and, since 2002, these standards have been also prescribed by RFI (*Italian Railway Network*) as a reference for the certification of products and electronic systems in safety in railway signalling. Some of these standards are listed below:

- **EN-50126 [9]:** Railway applications - The specification and demonstration of dependability, reliability, availability, maintainability and safety (RAMS).
- **EN-50129 [11]:** Railway applications - Safety related electronic systems.
- **EN-50128 [10]:** Railway applications - Software for railway control and protection systems.
- **EN-50159 [12]:** Railway applications - Communication, signalling and processing systems - Part 1: Safety related communication in closed transmission systems.

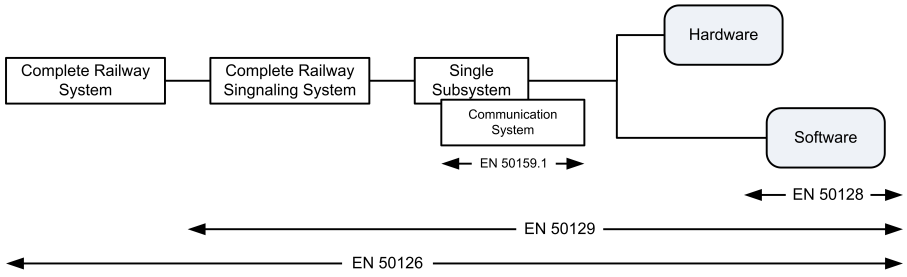


Figure 1.1: CENELEC norms scopes of application

The basic concept of the EN-50128 norm is the SSIL (Software Safety Integrity Level): the higher is the level of a system, the more serious are the consequences of a failure. Integrity levels range from 0 to 4, where 0 is the lower level, which refers to software with no effects on the safety of a system, and 4 is the maximum. These standards are in relation to each other as shown in Figure 1.1. The horizontal axis of the figure shows the scope of each standard, as is shown on the vertical progression from the most general to the most specific

standard.

The intention of CENELEC is to provide a model at European level in which the development of the safety-critical systems and their verification are oriented to totality of the *dependability* requirements. Dependability is defined as the property of a system to be "usable" by an human being, or a community, without the danger of unacceptable risks [3]. Dependability includes the following attributes:

- **Reliability:** is the capacity of a system to perform a required function, under certain conditions and for a specified period of time.
- **Availability:** is the ability of a system to perform a required function at a certain time or during a specified time interval, given the necessary resources.
- **Maintainability:** probability that for a given system unit can be carried out a given active maintenance action, during an interval of time, implemented through procedures and required means.
- **Safety:** absence of intolerable levels of risk of harm.

The development process of a system subject to the requirements RAMS (*Reliability - Availability - Maintainability - Safety*) is therefore composed of the following stages [59] :

- definition of the scope and objective of the system;
- analysis of the risk (e.g., calculation of the tolerable failure rate);
- design of the system functions. The requirements of each safety function defines the *integrity level* of the system;
- implementation;
- installation of the system;
- submission of the safety analysis produced during the development stages to the railway authority.

The safety requirements are central for this model of development: they guide the design and implementation of the system and are used as a reference when evaluating the model itself. The risk analysis provides the tools necessary to establish safety requirements; the process of risk analysis includes [59]:

- **Hazard analysis:** the *hazard* is defined as a risky situation that can lead to an *accident*. At this stage of the risk assessment it is necessary to identify the potential hazard, and reconstructs the sequence of events that can lead to them, in order to calculate the probability that these events will occur. This probability is classified by levels, ranging from *Incredible* (when it is assumed that a hazard can never occur) to *Frequent*.

- **Risk assessment:** in this phase are defined sequences of events leading from an hazard to an accident, which is classified according to severity, as well as the hazards. The severity varies from the maximum level of *Catastrophic*, where people are killed and wounded, to *Insignificant*, if only minor injuries and little damage to the system happen.

Once the two assessments above have been evaluated, it is determined what is the acceptable risk for each hazard, in accordance with the recommendations by railway authority.
Among the norms that are part of the CENELEC, EN 50128 is the one that specifies the procedures and technical requirements for the development of programmable electronic systems for the usage in railway control and protection applications. This norm applies only within the scope of the software (e.g., firmware, operating systems, applications) and the interaction between the software and the system.
The development of secure software also requires the application of some basic principles , including:

- modularity;

- verification activities at each stage of the development cycle;

- libraries and modules that are used must be verified;

- creation of clear documentation;

The system verification is an essential step: it is driven by the requirements, and aims to demonstrate that the system meets the safety and integrity requirements. Quantitative verification can not be carried out within the software, since the rate of failure and the contribution it gives to the occurrence of an accident cannot be quantified. For this reason the software is verified using qualitative techniques: the EN 50128 defines a life cycle for the definition of the specifications, the development and verification of software. At each stage of the cycle there is a verification process to ensure the compliance of the system to the level of integrity required. This life cycle is shown in Figure 1.2.

Figure 1.2: Phases of the life cycle of the software (CENELEC EN-50128)

The norm also lists tables in which in which the techniques that can (or must) be applied to ensure conformity of the system are related to the required level of integrity with the integrity level itself. An example of such a table is shown in

Figure 1.3. As you can see there are different priorities in the recommendations:

- **'M', mandatory:** indicates that the technique must be used for the given level of integrity;
- **'HR', highly recommended:** indicates that the technique should be used for the given level of integrity. If this technique is not used, this shall be justified and another equivalent technique shall be used in its place;
- **'R', recommended:** is a technique that can be used at the discretion of designers and verifiers;
- '-' : indicates that there are no recommendations in the use of the technique nor the reasons why it should not be adopted;
- **'NR', (not recommended):** indicates that the technique is not recommended for the given level of integrity. However, if this technique is used, it must be justified.

TECHNIQUE/MEASURE	Ref	SWSILO	SWSIL1	SWSIL2	SWSIL3	SWSIL4
1. Formal Methods including for example CCS, CSP, HOL, LOTOS, OBJ, Temporal Logic, VDM, Z and B	B.30	-	R	R	HR	HR
2. Semi-Formal Methods (incl. Statecharts, FSMs)	D.7	R	HR	HR	HR	HR
3. Structured. Methodology including for example JSD, MASCOT, SADT, SDL, SSADM and Yourdon.	B.60	R	HR	HR	HR	HR
4. Modular Approach	D.9	HR	M	M	M	M
5. Design and Coding Standards	D.1	HR	HR	HR	M	M
6. Analysable Programs	B.2	HR	HR	HR	HR	HR
7. Strongly Typed Programming Language	B.57	R	HR	HR	HR	HR

Figure 1.3: Example of relationship between techniques and levels of integrity

1.1.3 Quality of code according to standard CENELEC EN 50128

The norm CENELEC EN 50128 provides, among other things, information about the quality, the required structure, readability, traceability and testability of requirements [24]. In the table shown in Figure 1.3 it is possible to note how to

adopt standards in the coding is highly recommended for each type of system, regardless of the SIL, if not mandatory: it is highly recommended to provide a uniform layout of the code, as well as to strengthen the use of standard methods of safe design.
The *Coding Standards* are a set of rules and restrictions on a given programming language useful to avoid possible failures that can be caused by the use of that specific programming language. Among the provided rules are:

- justification for the use of a specific programming language;
- standard already available for a given programming language;
- restrictions to use to avoid failures;

Also the norm DO-178B [61], which is the reference document for determining the safety and reliability of software in avionics, defines the *Software Code Standards* as concerning the code formatting and the conventions used for the names of variables, functions, and so on, and this is further evidence that the code quality is critical in environments in which software safety-critical related systems have to be developed: in fact the adoption of a common and consistent coding style facilitates the understanding and maintenance of the code, especially if it is developed by multiple people in co-operation.

1.2 Problem statement

This dissertation is the result of a three years reasearch activity and collaboration with GETS, started at the end of 2009, aiming at addressing the following:

Problem Statement
Defining and implementing a methodology for the adoption of formal methods and code generation technology in the development of safety-critical systems by a railway signalling manufacturer.

During the research, the problem statement has been decomposed into the following sub-goals:

1. Finalize the introduction of the Model-Based Design in the development process with a special focus on the validation of the translation process and the verification process. The generated code correctness is fundamental, CENELEC EN 50128 norm [10] asks for a certified or proven-in-use translator. Since Simulink/Stateflow have not this property, a strategy has to be defined in order to ensure that the code behavior is compliant to the model behaviour, and no additional improper functions are added during the code synthesis phase.

2. Integrate the new formal development techinque in the process, also in the early phases of development, such as requirements specification and architecture specification. The introduction of new technologies in an established process is not straightforward: the process structure shall be adjusted and shall be maintened coherent even if changes are applied. For GETS, which operates in a safety-critical systems development context, this is fundamental, since its products have to be validated according to the prescripted normatives.

CHAPTER 2

Model Based Design and Code Generation

Model-Based Design (MBD) [71] has been widely accepted and adopted by industries. The basic concept of the MBD is that the whole software development process aims to produce graphical model abstractions, from which to derive an implementation, manually or automatically. There are tools which support the possibility to simulate and test directly on the model (e.g., Simulink/Stateflow [82], SCADE [19], Scilab/Scicos [68]) before the implementation is actually derived. In its objective, i.e., to detect design defects before the project is deployed, the MBD approach is not different from the formal methods, but the latter are perceived more difficult to use than the former: MBD is closer to the needs of developers, that consider the graphical approach more intuitive than a formal description.

Simulink/Stateflow has been elected as *de-facto* standard for modeling and code generation for safety industries, surpassing other tools such as SCADE, based on Lustre synchronous language, Scilab/Scicos, an open source platform for modelling and simulating control systems, ASCET [20] and AutoFocus [38], both using block notations for the representation of distributed processes and embedding advanced verification capabilities. Simulink/Stateflow provides a lot of built-in blocks and libraries, and its intuitive state machine graphical editor (Stateflow)

allows the developers to build prototypes and simulate directly on them in a very intuitive way.
In this chapter it is briefly introduced the Model-Based Design approach and it is presented the Simulink/Stateflow toolsuite that has been chosen by the company in order to apply the MBD development paradigm to the development of safety-critical railway signalling systems.

2.1 Basic concepts of Model-Based Design

The use of models for the development of complex systems is now an established practice in many engineering disciplines, such as the construction of complex buildings such as bridges, or, in the mechanical domain, cars. The models help in understanding the problem, and thanks to a high level of abstraction also help in finding the solution. It is obvious, therefore, that this approach to development has recently been proposed for the creation of software, often considered among the most complex engineering systems. The Model Based Design is based on the paradigm that the main objective, and the product, of the software development are the models instead of code listings [71]. The biggest advantage that is derived from this approach is the fact that the models are expressed using concepts, which are less restrictive than the technologies that must be employed for implementation, and are closer to the problem domain that is being analyzed. This translates into ease of specification, understanding and maintenance of the designed system. Another important advantage is the independence of the chosen programming language: the models in fact possess a sufficiently high level of abstraction to be able to decouple from the implementation choices.
To exploit the full potential of MBD is necessary to:

- generate the software completely from the model. This approach is relatively recent, as earlier models were only used to generate the software structure (*skeleton*) or code fragments. The automatic code generation assumes with respect to the programming languages the same role that the third generation of languages has taken with respect to the assembly language.

- automatically verify the models (e.g., by executing the model it is simulated the behavior of the system).

Therefore, the models are used for both the generation of the code and, in a certain way, for the verification of the same. Figure 2.1 schematically shows the difference between an approach that uses the model as a basis for development-level documentation (left) and MBD approach with complete generation of code (right) [46].

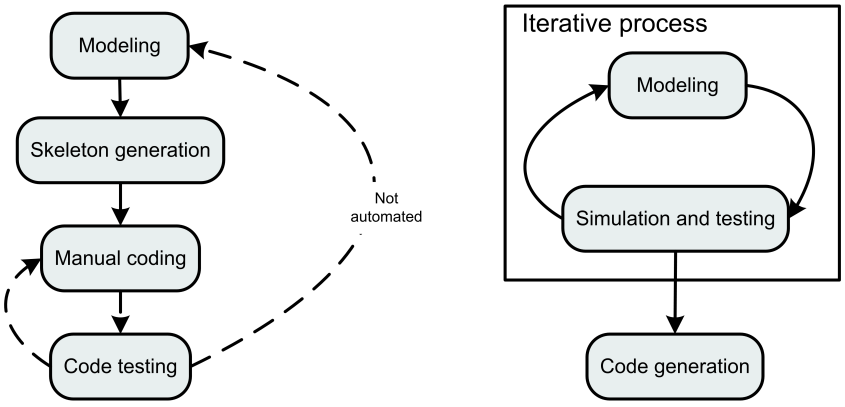


Figure 2.1: The MBD approach with code generation

In order for the MBD approach to be useful and effective, the models constructed must have five key features [71]:

Abstraction is the most important feature. A model is a reduced-scale translation of the system that is meant to represent. If you go to hide, or eliminate, the details that are irrelevant to a given point of view, the model allows to understand more easily the essence of the system more easily. Given that the required systems are becoming increasingly sophisticated, abstraction is the only method that allows to deal with the increasing complexity.

Understandability: compared to the directives used in the code listings, which require an understanding of often extremely intricate syntactic rules, the model allows to have a a rather clear overview of the system. Abstraction

is not enough, but it is necessary to take into account, during the process of modeling, this desirable property that the model should have.

Accuracy: a model should provide a representation consistent with the capabilities of the system that it purports to represent.

Predictability: it shall be possible, through the use of a model, to predict the properties of interest of the system, both through the execution of the model (i.e., the simulation of the system) than through some kind of formal analysis.

Convenience: the last property that the model must have is that of being "economic": this means that it must be cheaper to build and analyze the model rather than the modeled system.

Given the characteristics listed, it is essential to take into account the quality of the models that are being created.

2.2 The importance of the quality of the models

To ensure that the MBD approach is also applicable in the development of railway protection systems, it is essential that the code generators used provide a translation from model to code of a certain quality, that complies with the regulations in force in this field.
Given what has been said about it, it is easy to see how the models are closely related to the system they represent, and as a result the quality of the models is crucial to the quality of the produced software. This has given rise to the need to transfer to the models a series of checks that were previously performed on the code listings: in fact, the MBD approach allows the application of various types of analysis directly the model (e.g., the static analysis and simulation), reducing the number of checks carried on directly on the code, which are the most expensive ones. Many companies in various sectors, such as rail, automotive and aerospace industries, which have adopted the MBD approach, have created rules and verification techniques that, when applied to models, allow to understand

if the code that will be generated automatically from those models will meet expected directives, which are defined to improve quality, increase safety and efficiency.

2.3 Brief introduction to Simulink / Stateflow modeling tools

In this section it will be briefly introduced the Simulink/Stateflow [82] modeling suite, which increased its popularity in the last years also thanks to its usage in the Model-Based Design by major companies working in safety-critical contexts (e.g., Airbus [78], Alstom [79], NASA [84]).
This introduction will give an overview on principal components of Stateflow tool, starting from the FSM, modeled through Stateflow Graphic Editor, to the Automatic Code Generation functionalities given by the Stateflow Coder.

2.3.1 Introduction

Stateflow is a tool for modeling by means of Harel state machines [36]. The tool is fully integrated in Simulink and interacts with it in the creation of models with a high level of detail: however Simulink is suitable for the modeling of dynamic systems, dealing with the dynamics for continuous-time and discrete-time systems, while Stateflow provides a suitable environment for the development of models aimed at the realization of logical algorithms and event-driven reactive systems.

Domain

Stateflow is today widely used in many industries, among which stand the automotive one and aerospace one. The joint use of Simulink and Stateflow allows the creation of embedded control systems. Simulink is used for the implementation of numerical algorithms, while Stateflow is better suited to the creation

of the control logic of reactive systems, i.e., systems that change their state in response to an event. The reactive systems are thus modeled as state machines [24] [44].
Stateflow is extremely useful for the design of:

- **Functioning logic:** control of the system behavior according to its functioning mode. The system under development can thus assume a limited number of functioning states.
- **Supervisory logic:** flow-chart design. It determines how the system behaves in reaction to events or conditions.
- **Scheduling:** scheduling and timing of activities.
- **Errors management:** supervision over the process of identification, isolation and resolution of error conditions in the system.

Stateflow basic concepts

Stateflow is based on the theory of state machines enunciated by David Harel in the late '80s [36], and adopted by the UML standard. Harel introduced an enhanced version of the state machines, retaining the graphical formalism but introducing innovative elements that allow extremely flexible modeling and a high level of detail, impossible to achieve with traditional state machines.
The innovations concern:

- deterministic automata;
- hierarchical approach (states can have different priorities during execution);
- parallelism (more than one state active at the same time);
- new semantics;

In this context, a finite state machine is defined as the representation of a reactive system, driven by events. In such a system, the transition from one operative mode, or state, to another is accomplished when certain conditions occur.

Generally, the finite state machines are represented by their graphs of components, states, and the connections between them, the transitions: the behavior of the whole designed system is provided in terms of transitions between states; the active state at a given time is determined precisely by the path that goes from the initial state and moves along transitions triggered by conditions.

2.3.2 Stateflow

Stateflow allows to build finite state machines that can be used to simulate the behavior of a system, or to generate code that will be used for embedded systems (e.g., control systems). Below are presented the main components available in Stateflow to create such machines.

Stateflow elements

States

The *state* is one of the fundamental components of Stateflow, and describes a mode of operation of an event-driven system. The fact that a state is active or not changes dynamically based on events that have occurred and the conditions that have been met during the *execution* of the FSM.
Every state has a parent state, even if the Stateflow diagram is composed of a single state: in this case the parent is the state diagram itself, which contains the state. It is possible to insert states in other states, so as to form a *hierarchy*. Each state has a name that uniquely identifies it among those of the same hierarchical level. In addition, in the label of a state, other than the name, can be specified a number of actions that must be performed at particular times of execution:

- **entry (en):** the action that follows this keyword is performed by the system every time that the state is activated, i.e., an event brings the FSM into that specific mode of operation;
- **during (du):** the action that follows this keyword is performed until the system has this specific mode of operation. Once the system exits from the mode of operation related to this state, the action is no more performed;

- **exit (ex):** the action that follows this keyword is performed when the system changes mode of operation, i.e., exits from the state;
- **on *"event"*:** the action that follows this keyword is performed when the specified "event" occurs.

An example of a state with specified actions is represented in Fig. 2.2.

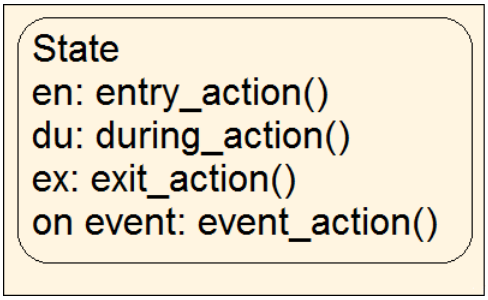


Figure 2.2: Actions specified in the label of a state

Stateflow provides two types of states that can be used for the realization of the finite state machine which models the system:

- **Parallel states (AND)**, shown in Fig. 2.3, that can be simultaneously active at a given instant of execution. The picture shows that both states are highlighted in blue, that indicates that a state is active at a given moment of time.

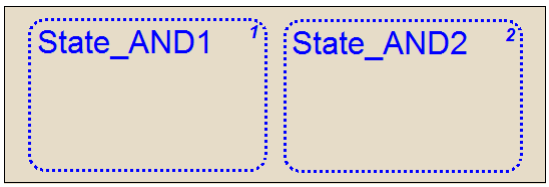


Figure 2.3: Parallel states execution

- **Exclusive states (OR)**, shown in figure 2.4, which can not be active simultaneously. Also in this case the pciture shows how only a state during

the execution of the two is active, as only one of them is highlighted in blue.

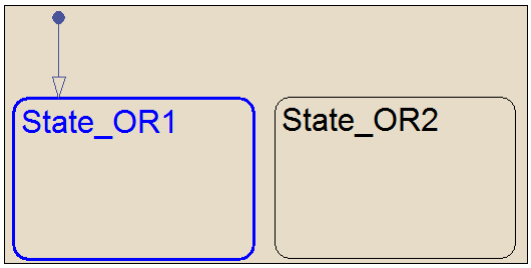


Figure 2.4: Exclusive states execution

Transitions

A transition is a graphical object that, in most cases, connects an object to another: the transition is characterized by a source object and a target object, that not necessarily are states. The label on a transition describes the condition that enables passing from the source to the destination object.
A transition can thus connects:

- two states;
- a state and a junction;
- two junctions;
- a state and a box object;
- a junction and a box object;

It is important to note that the transitions have a direction. An unlabeled transition that connects two states is shown in Fig. 2.5.

Conditions

The conditions are Boolean expressions that specify, with their value, if a transition should be triggered or not, given that the source state of the transition is

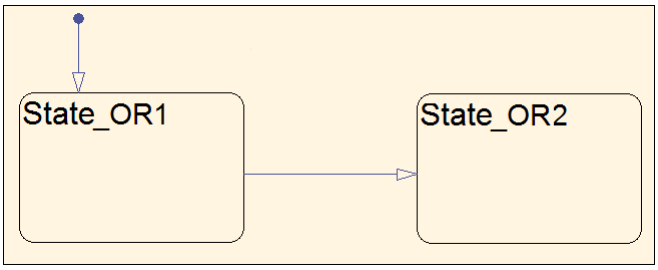


Figure 2.5: A transition between two states

active at the time when the condition becomes true. Wanting to make a practical example, we may consider the change of gear of an engine, which occurs when the speed exceeds a certain threshold. This can be represented with the chart shown in Fig. 2.6. The condition is indicated on the label of the transition and enclosed in brackets. The figure also shows the hierarchy that may exist between the states: the state *Gear* contains the two states *First* and *Second*, making the state *Gear* the parent of the other two states.
Gear_shift is instead another type of Stateflow construct, i.e. an *Event*: this further type of objects that Stateflow provides to modelers will be described below, in a dedicated section.

Actions

The actions, when specified in the label of a transition, can belongs to two classes:

- actions related to the condition, which are executed when the corresponding condition is true at the moment of the transition;
- actions related to the transition, which are executed when the transition is enabled, without dependence on conditions.

Both types of actions are represented in Fig. 2.7.

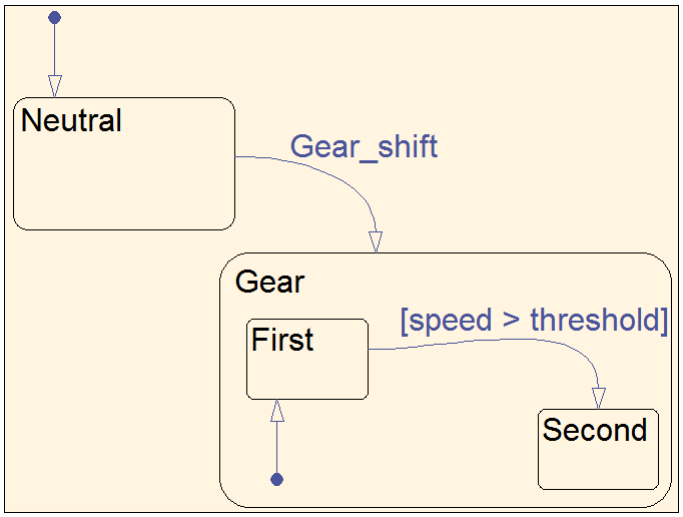


Figure 2.6: Example of a condition for the activation of a transition

Default transition

The default transition is a transition that specifies which particular state must be active when there is ambiguity between two or more exclusive states at the same hierarchical level. Fig. 2.8 shows an example of the usage of the default transition: when the machine is turned on and reaches the state *Lights*, the operation mode is placed on the state *Off*, which is connected to the default transition, as long as the event "Switch_on" occurs. From the figure it can be noticed that the default transition does not have a source object, but only a target object.

Data

The data are non-graphical objects that store numeric values used in Stateflow charts; not having a corresponding graphic object, data are not directly represented on the chart, but can be accessed through the user interface Model Explorer, shown in Figure 2.9. Each data has its own scope, which defines whether the data is:

- local to the Stateflow diagram;

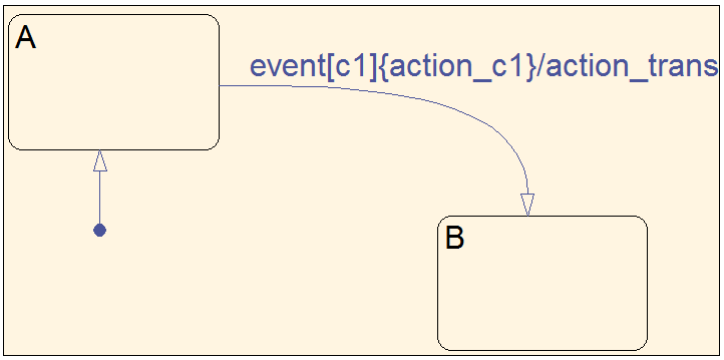


Figure 2.7: Actions related to a transition

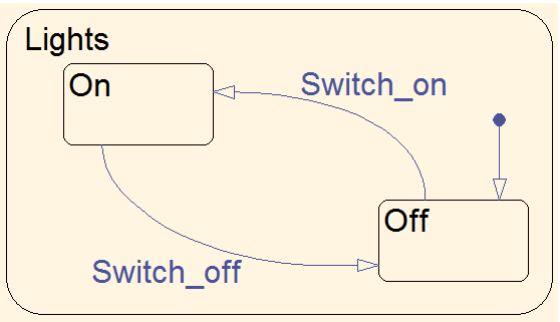


Figure 2.8: An example of using the default transition

- an output that goes from the Stateflow diagram to Simulink diagram of which the first is as a component;
- an input from the Simulink diagram;
- data defined in the MATLAB workspace;
- constants;

From the image it is possible to note that Model Explorer provides numerous information regarding the data, among which, as already mentioned, the scope, the data type and the name of the data.

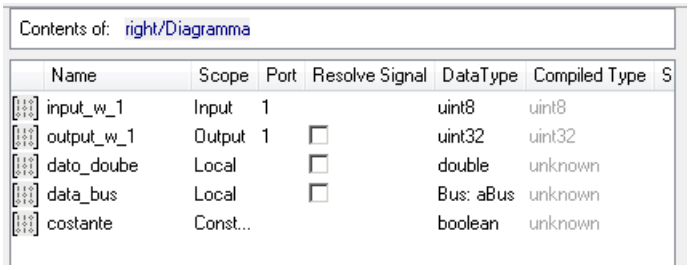


Figure 2.9: Displaying Data in a Stateflow diagram using Model Explorer

The data can also be aggregated into unique objects, which are called *Bus*, which can be compared, for example, to the structures in the C programming language.

Events

The events are non-graphical objects, similarly to data. Events guide the execution of the Stateflow diagram, and have a function similar to the switches. Events can be managed through the Model Explorer; events can also be created at any level of the hierarchy, and possess properties such as scope, which defines whether an event is:

- local to the Stateflow diagram;
- an output that goes from the Stateflow diagram to Simulink diagram of which the first is as a component;
- an input from the Simulink diagram;

The events can also be found on the labels of transitions: in a similar way as applies to the conditions, the transition becomes active when the event that is specified on the label occurs.

Junctions

The junctions are decision points in the system. A junction is a graphical object that simplifies Stateflow diagrams representations, implementing constructs such as *if-then-else*, shown in Fig. 2.10. The image also shows the code corresponding

to the structure.
The junctions are crucial for the creation of Flowchart (functions), which will be presented in the following paragraphs.
The transitions that are connected to the junction are called *segments*, and have the same properties as any other transition.

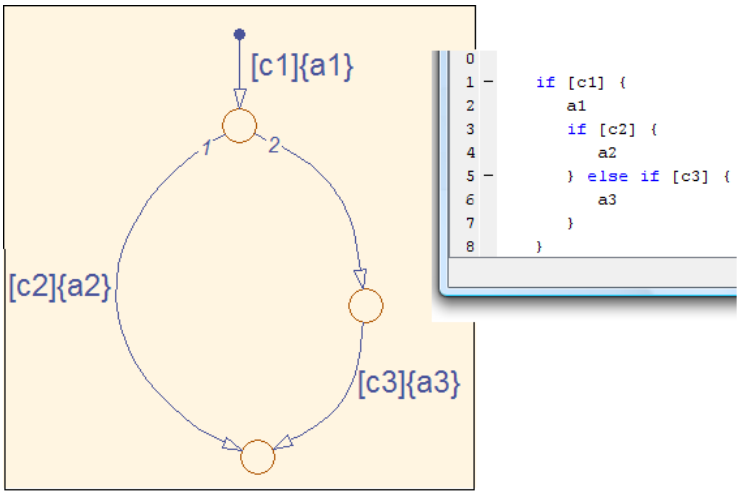


Figure 2.10: If-then-else implemented through junction

Graphical functions

The graphical functions (or flowchart) are functions defined graphically by a flow chart, and add expressive power to the language of the actions defined in Stateflow. The functions are formed by transitions and junctions interconnected, as shown in the example in Fig. 2.11. A function can have arguments and can provide return values: in the case depicted by Fig. 2.11, the input arguments are a and b, and the return value z is given by the sum of the squares of the two arguments.

2.3.3 Graphic Stateflow Editor: an example

Simulink/Stateflow toolsuite can be used, as already introduced, for modeling a system that requires a certain logic of operation. In this section it is presented an example of the modeling through Stateflow of a controller for a system with

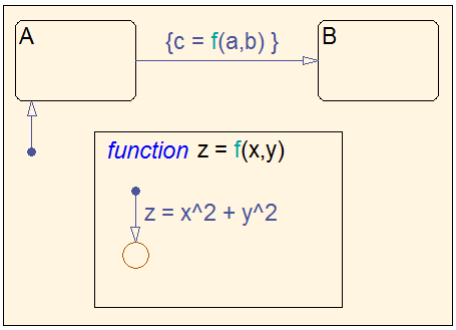


Figure 2.11: Example of use of a function

two fans for cooling.
In Fig. 2.12 is shown the diagram representing the system, which decides, on the basis of the detected temperature and provided input, if none, one or two fans shall be activated. The Stateflow chart is integrated into a Simulink model, not shown, which implements additional components forming a larger system.

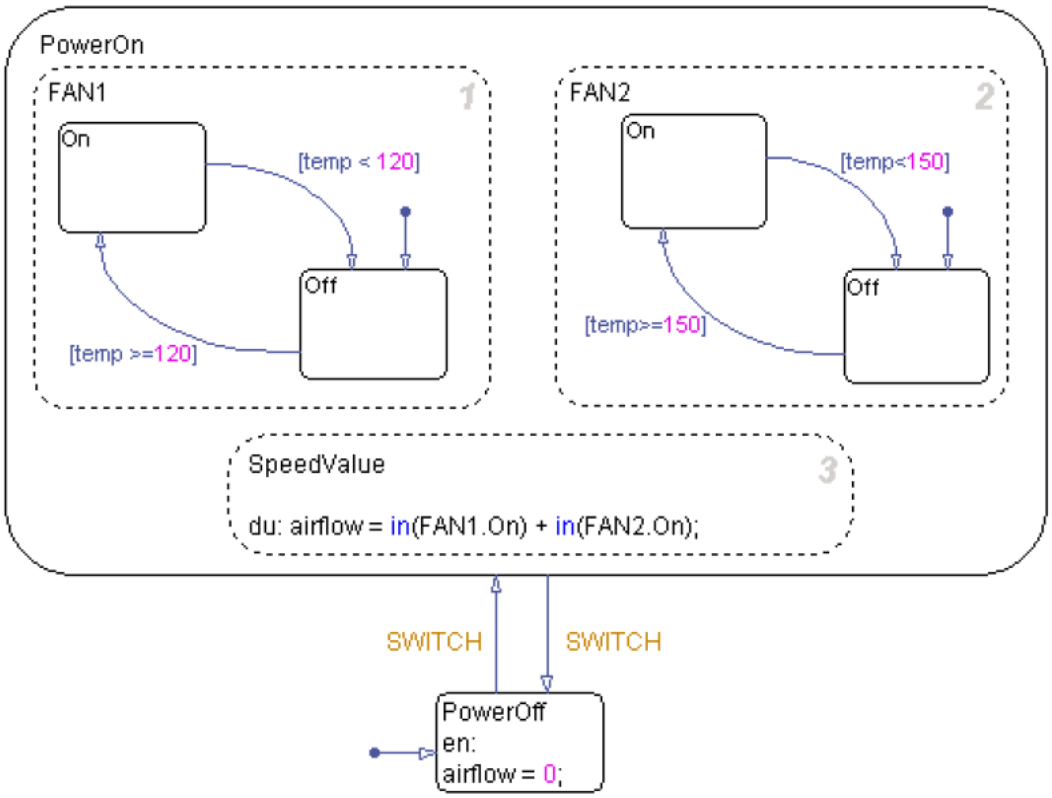


Figure 2.12: Example of Stateflow chart that implements a controller

The diagram is rather simple: it includes many of the components seen in the previous paragraphs; the system is composed, at the top level of the hierarchy, by two exclusive (OR) states; once started, the system goes in the state corresponding to the operating mode in which the system is off (*PowerOff*), which is achieved by the default transition. If the event *SWITCH* occurs, the system goes in the opposite state with respect to the current one (therefore will turn on if it was off, and vice-versa). The state *PowerOn* is composed of three substates, and all of them execute in parallel (AND). The states are:

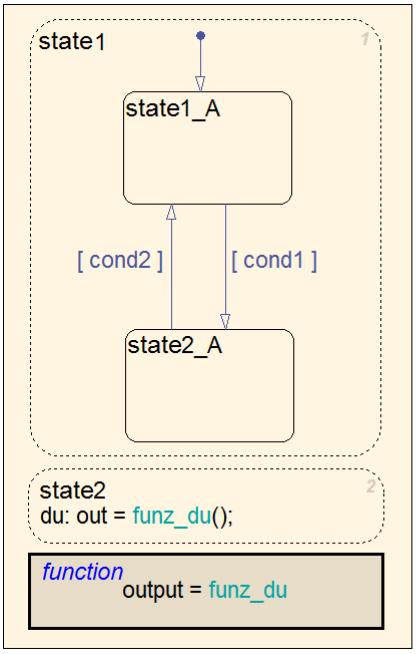
- *SpeedValue*, which takes care of calculating at each instant (thanks to the keywords *du* in the state label) the air flow output from the diagram (*airflow*);
- *FAN1*, which implements the control logic of the first fan: when the tem-

perature exceeds 120 degrees, the fan turns on, otherwise it is turned off;

- *FAN2*, which implements the control logic of the second fan: in this case, the threshold temperature is 150 degrees.

Stateflow Coder: code generation starting from a model

As already introduced, Stateflow provides the ability to generate code from models, thanks to the instrument Stateflow Coder [81]. Stateflow Coder is able to generate code from all objects and semantics Stateflow, allowing to realize standalone applications or code that will be inserted into existing applications.
Once you have created a state machine using Stateflow, you can generate code using the Model Explorer, a user interface that provides a variety of information on a Simulink model and its components, including the Stateflow chart. The language in which code is generated is the ANSI-C [2] standard issued in 1990 by the American National Standards Institute, and extremely used, along with Ada [40], in many industrial fields that require safety standards.
To complete the overview is shown in Fig. 2.13, a code listing generated from a simple Stateflow diagram, also shown in the same figure.




```
typedef struct {
  uint32_T out;
} SFc1_testOutputDataStruct;

typedef struct {
  boolean_T cond1;
  boolean_T cond2;
} SFc1_testInputDataStruct;
```

```
void c1_test(SFc1_testInputDataStruct *chartInputData,
     SFc1_testOutputDataStruct *chartOutputData) {
  if(chartInstance.is_active_c1_test == 0) {
    chartInstance.is_active_c1_test = 1U;
    chartInstance.is_active_state1 = 1U;
    chartInstance.is_state1 = (uint8_T)IN_state1_A;
    chartInstance.is_active_state2 = 1U;
  } else {
    switch(chartInstance.is_state1) {
     case IN_state1_A:
      if(chartInputData->cond1) {
        chartInstance.is_state1 = (uint8_T)IN_state2_A;
      }
      break;
     case IN_state2_A:
      if(chartInputData->cond2) {
        chartInstance.is_state1 = (uint8_T)IN_state1_A;
      }
      break;
     default:
      chartInstance.is_state1 = (uint8_T)IN_state1_A;
      break;
    }
    chartOutputData->out = funz_du(chartInputData, chartOutputData);
  }
}
```

Figure 2.13: Code generated by a diagram consisting of multiple states, both parallel and exclusive

CHAPTER 3

MBD in the GETS Safety Critical Systems Development

General Electric Transportation Systems develops embedded platforms for railway signalling systems and, inside a long-term effort for introducing formal methods to enforce product safety, employed modelling first for the development of prototypes [6] and afterwards for requirements formalization and automatic code generation [24]. Within the new development context also the verification and validation activities have experienced an evolution toward a more formal approach. In particular, the code-based unit testing process guided by structural coverage objectives, which was previously used by the company to detect errors in the software before integration, has been completely restructured to address the new model-based paradigm. The process refactoring has been driven by three main reasons:

- the traditional approach based on exercising the code behaviour resulted in being too costly to be applied to a code that saw a size increment of four times for the same project within two years. This fast growth was partly due to the increase of the actual projects size and partly to the code generators that, as known, produce more redundant code than the one that could be produced by manual editing;
- when an automatic tool is used to translate from a model to software, it

has to be ensured that the latter is actually compliant with the intended behaviour expressed by the model;

- unit testing alone, whether model-based or code-based, cannot cover all the possible behaviours of the code in terms of control flow and data flow. Most notably, it lacks in detecting all those runtime errors, such as division by zero and buffer overflow, that might occur only with particular data sets.

The restructured unit level verification process is based on two phases, namely model-based testing and static analysis by means of abstract interpretation. The first phase is used to exercise the functional behaviour of models and code, and, at the same time, to ensure that the synthesized code conforms to the model behaviour. The second phase is used to ensure that the code is free from runtime errors. Unit level verification costs were in the end reduced of about 70%, while decreasing the man/hours required for bug detection and correction. During the research, particular focus has been given to the strategies followed to address formal weaknesses and certification issues of the adopted tool-suite. In previous projects, experimentation with the code generator led to the definition of an internal set of modeling rules in the form of an extension of the MAAB guidelines [44], a stable and widely accepted standard developed by automotive companies. The guidelines concept is presented in §3.1.1. Concerning verification of models and generated code, an enhancement of the two-phase approach has been adopted: control-flow and functional properties have been verified through model-based testing, presented in §3.2, while static analysis by means of abstract interpretation (this issue has been studied wthin another PhD research activity [34]) has been used to check data-flow properties. The model-based testing activity has been performed through a code validation framework that executes the same test cases both at model level and at code level. The framework automatically verifies consistency of the test results for each model unit. This idea basically settles the problem of having a qualified code generator, since certification of conformity can be ensured each time the code is synthesized from a model.

3.1 MBD in the development process

The MBD process adopted represents an application of model-based practices to a V based life-cycle, and it is the result of the collaboration between GETS and University of Florence [28]. Four phases are considered as the core of the system development: architecture, design, module verification and system integration/verification. Also formal verification has been introduced as experimental verification activity, although not yet consolidated into the currently practiced development process. Architecture and design activities concerning the project have been reported in [25], while the verification steps represent an enhancement of the approach presented in [8].
As stated before, the tool that GETS uses for the creation of the models is Stateflow, by The MathWorks. The choice of using this tool rather than others was dictated by a number of considerations accrued as a result of numerous assessments of the potential environmental Stateflow, many of them carried out in collaboration with the University of Florence [6].

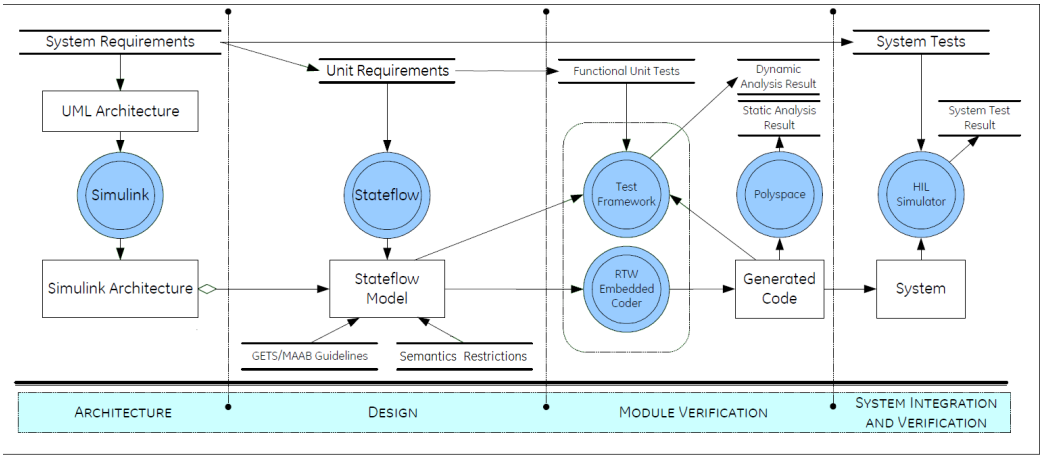


Figure 3.1: Process Overview

Fig. 3.1 summarizes the overall process structure. Starting from system requirements and using domain knowledge, a functional architecture in the form of a UML component diagram is defined consisting of independent functional units. According to this decomposition, system requirements are partitioned into disjoint sets of unit requirements to be apportioned to the functions. The UML

architecture is then translated into a Simulink architecture and the unit requirements are formalized in terms of Stateflow finite-state automata.
Stateflow models are designed using a safe subset of the language in order to ensure proper code synthesis [28]. Already in previous projects an extension of the MAAB guidelines was used, aimed at enhancing the readability, maintainability and structuring of the code. Additional restrictions have been introduced to further constrain the Stateflow semantics to an unambiguous set of constructs (see §3.1.1). Concerning code generation, the company adopted the more customizable RTW Embedded Coder in place of the previously used Stateflow Coder: the transition was basically painless, since all the modeling rules developed for the previous tool resulted in being applicable also for the new one.
Unit tests have been defined in the form of scenarios at the Stateflow model level (§3.2). To this end, an internally developed framework has been used that automatically executes the test suite on the Stateflow automaton and on the generated code to ensure functional coherence between model and software behavior. The confidence on the correctness of the generated code is increased with the Polyspace tool for abstract interpretation [34], which completes the unit-level verification activities. Finally, system tests are performed on an ad-hoc train simulator with hardware in the loop.
During the research, formal verification (§3.2.2) has been experimented at module-level using Simulink Design Verifier (DV), a property proving engine that uses Simulink/Stateflow both as modeling and as property representation language. Unit requirements have been translated into Simulink formulae against which Stateflow models have been verified.

3.1.1 Modeling guidelines

The quest for ever higher quality models, geared to increase the quality of the generated code as required by the specification as CENELEC EN 50128, has led GETS to the creation of corporate guidelines, called *Istruzioni di Ente*, which must be applied in modeling stage by the designers, in order to achieve the above mentioned qualities.
As stated before, the use of modeling guidelines can bring many improvements both in the model and in the generated code starting from the model itself. It

is also important to adopt a standard that describes the modeling rules, so that they can be understood by everyone involved in creating diagrams and anyone concerned to verify them. One of the available templates for the statement and the description of the guidelines is created by the *MathWorks Automotive Advisory Board* (MAAB, [44]) for Simulink and Stateflow models and used in the document *Control Algorithm Modeling Guidelines Using MATLAB, Simulink and Stateflow* [44]; the template, adopted also for the *Istruzioni di Ente* GETS and shown in Figure 3.2, is thus formed:

ID: Titolo	
Priorità	
Visibilità	
Versione MATLAB®	
Prerequisiti	
Descrizione	
Obiettivo	☐ Leggibilità ☐ Verifica e Validazione ☐ Workflow ☐ Generazione del Codice ☐ Simulazione
Ultimo Cambiamento	

Figure 3.2: Template for the statement and the description of guidelines

- **ID:** the ID of a rule must be composed of two lowercase letters and four numbers separated by an underscore symbol (i.e.,"_"). An ID is unique, and once assigned to a guideline should not be changed anymore.
- **Title:** the title should provide a brief description of the rule, and which aspects of the model it covers.
- **Priority:** priority has a dual function, that indicates the importance of the rule and determine the consequences of violations. Must be selected from the following:

  - *Mandatory:* are the rules considered essential, and those for which the company guarantees full compliance of its models. If violated, the model may not work according to specification. If the guideline is ignored, both in design and in testing phase, the reason must be documented.
  - *Strongly recommended:* a rule that has this priority must be respected as long as there are corporate policies that prevent or limit its applicability. It is not necessary that the models conform to these rules 100%, therefore a violation of the same translates, for example, in a deterioration of the model or problems regarding reusability and maintainability, but does not affect the operation of the system.
  - *Recommended:* the guidelines that this priority is used to improve the look and feel of the model, but they are absolutely critical with respect to the operation or other important aspects.

- **Visibility:** it represents the scope of the rule, i.e., if the rule has been integrated from external rules (e.g., MAAB [21]) or it has been developed internally to the company.

- **MATLAB version:** represents the version for which the rule applies. If a guideline contains the special features that limit its application in all versions of the software, this should be specified in this field.

- **Prerequisites:** this field points to those rules that are a prerequisite for the rule in question, in the sense that in order to satisfy the rule in question, the model should first satisfy the prerequisite ones.

- **Description:** Here is a description of the rule in natural language. If necessary, images and tables can be added.

- **Objective:** the goal is the reason why the guideline was designed:

  - *Readability:* it concerns the documentation, the appearance of the model (and hence the visual comprehension), the design of usable interfaces;

  - *Workflow:* it concerns the ease of development, maintainability, reuse of model, and portability;
  - *Simulation:* a rule that has this target is proposed to increase the execution speed of the simulation of the system, and to decrease the required memory;
  - *Verification and validation:* conformity with a rule that has this goal improves the traceability of requirements and testing;
  - *Code Generation:* it concerns the quality and the robustness of the generated code.

- **Last change:** in this field is specified in which version of the document has been made the last change to the guideline.

3.2 Generated code verification and MBT

Traditionally GETS has used, as the main approach to verification of code units, white-box testing based on path coverage. This approach has revealed to be almost unfeasible due to the high structural complexity of the automatically generated code. A two phase verification process [8] was defined to address this shortcoming: the first phase implements model-based testing to verify the functional requirements coverage, the second phase employs abstract interpretation [18] to statically enforce runtime errors detection. The first phase has been later improved to obtain an implicit validation of the code generator as well.

The adopted approach for MBT comprises two steps. The first one is a form of model/code back-to-back (B2B) testing [76], where both the model and the related generated code are tested using the same stimuli as inputs, and the numerical results obtained as output are checked for equivalence. The second one consists of an additional evaluation to grant the absence of unwanted and unexpected behaviours introduced by the model-to-code translation process. This evaluation is basically the comparison of the measures of structural coverage reached on both the code and the model. The two phases could be seen respectively as a duplicated black-box testing (output comparison) and a duplicated

white box testing (coverage comparison).
The B2B testing step is depicted in Fig. 3.3 (a). First, a Stateflow model is created starting from the unit requirements. This model represents the behavioural view of the system. In order to verify if this behaviour is compliant with the specification, a test suite for the model is manually derived from the specification, according to the full requirement coverage criterion (i.e., at least one test for each functional requirement). The outputs given by the simulation with the given test data are visually checked to assess those behaviours which do not comply to the specifications.
A tool called *Test Observer* was developed and used to automatically translate the functional unit tests written for the Stateflow model into an appropriate form to be used as tests for the generated code. Test Observer records both input and output sequences of the model during the simulation in the form of Simulink time series, that are Simulink data objects composed by pairs (time, value), and then translates the time-series into given input/expected outputs matrices for the generated module. C code is then generated starting from the model, using RTW Embedded Coder. For each test case another tool, called *Test Integrator*, produces an executable file that embeds the given input/expected output matrices, together with a set of functions to check if the output of the generated code matches the expected output. If the code executes without errors, then it can be stated that, for the given functional unit test, the generated code shows the same behaviour of the model [33].
The B2B testing task has been improved during the research activity and integrated in a new framework, called 2M-TVF (Matlab Model to Validation Framework), whose workflow is depicted in Fig. 3.3 (b). The 2M-TVF framework offers functionalities of validation of the code generation process and of the generated code itself, and it is further detailed in the next paragraphs.

3.2.1 2M-TVF and the translation validation

As stated in the previous chapters, products traditionally developed by GETS, like any railway signalling application developed for Europe, shall comply with the CENELEC standards [10]. The CENELEC EN 50128 considers two alternative strategies to assure the correctness of the tools that produce the code that is

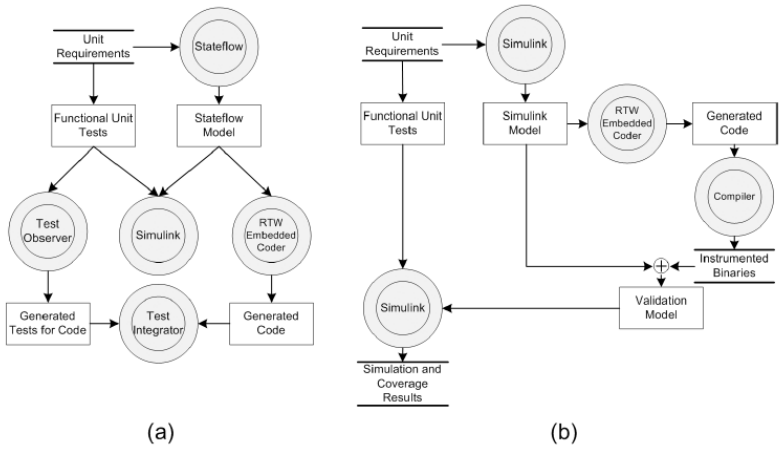


Figure 3.3: Approaches of the automation of MBT

finally embedded in the product, such as compilers or automatic code generators: the first one requires the code generator to be validated, and the second one is based on the so called *proven in use* property. However, both strategies could not be applied for the adopted code generator: the validation of the generator is unfeasible, since the source code is proprietary and no information is given about its development process; also the translator could not be considered as a *proven in use* tool. Furthermore, the creation of a specialized tool, and its validation, is not in line with the company strategy, which completely resorts to the usage of commercial tools.

The problem has been addressed for the first time in the context of the Metrô Rio project[1], using an approach that is inspired to the one presented in [17], called Translation Validation[2]: this approach is not focused on the code generator itself, but on the inputs and the outputs of code generation process and on their comparison. The validation of the generated code is performed through two phases:

1. Dynamic testing (running the model through simulation and generated

[1]Metrô Rio is a project started in 2008 that concerns an ATP (Automatic Train Protection) platform for the metro of Rio de Janeiro

[2]Translation Validation refers to a process that involves the comparison between code and models behavior as a model-based conformance testing. The term Translation Validation has been previously used in literature to address a similar problem in [60]

code and compiled) and subsequent comparison of the results obtained. This phase is analogous to the B2B phase in its first form (see §3.2). The tests shall be extensive, defined starting from the modeled requirements, with a 100% requirements coverage. In the first phase of the validation of the generated code, a functional/black box testing is performed, where both model and code are stimulated with the same inputs, and their outputs are compared for equality. If there are differences between the model outputs and the code outputs, they shall be assessed.

2. Additional assessments to ensure the absence of unexpected functions introduced during the translation process model code (e.g., comparison of the measurements of the model and code coverage). In order to perform this evaluation, it is necessary to use comparable metrics for model and code coverage [4]. In our case we chose to use decision coverage for the model and branch coverage for the code[3]. Since a branch is the outcome of a decision, branch coverage is defined as the ratio between the number of decision outcomes that have been exercised by a test suite and the total number of decisions in the code. The choice of those metrics is due to the fact that the CENELEC EN 50128 requires at least statement coverage, that is automatically implied when full branch coverage is achieved; furthermore, since no continuous dynamic Simulink blocks are used in our modeling approach, the decision for those metrics is appropriate. As stated in [17], if the code coverage obtained after test execution is less than the model coverage, then some unwanted functionality might have been introduced by the translator.

2M-TVF, which stands for Matlab Model Translation Validation Framework, is a framework that has been developed to perform the model-based testing and the code validation process. The framework works under the Simulink environment, and in order to carry out the validation process, allows for the definition of a validation model that includes both the Matlab model to be tested and the related generated code, embedded in a Simulink block. The code is previously

[3]Decision coverage and branch coverage are synonymous. It was decided to maintain both definitions since the former is used in code-testing context, while the latter is used in model-testing contexts

instrumented to permit the evaluation of the coverage after the tests execution, and then it is compiled and linked to obtain the executable.
The validation of the translation process is divided into two phases:

The approach described above is shown schematically in Fig. 3.4.

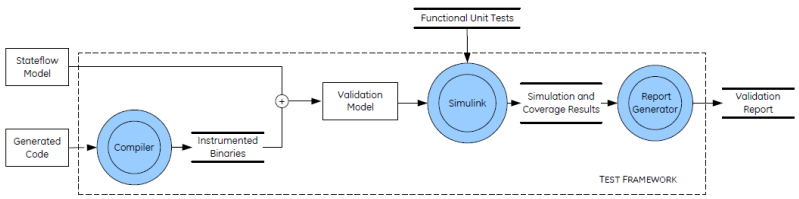


Figure 3.4: Stages of the process validation of the code generation

To be able to perform the described evaluations is thus necessary to have a well-structured model, which is corresponding to what is reported by the requirements, and a series of tests to be run on the model. It is also necessary to choose a metric of adequate coverage, both for the model and, in congruence, for the code.
After performing the tests, any difference in the results, be it in the numerical comparison of the outputs or in the coverage of the model and the code must be justified by identifying any systematic behavior of the code generator or real errors in the translation from model to code.
It is important to emphasize that the process of validation concerns the process of generating code and the generated code, but not the generator itself: to demonstrate the ability of the generator to correctly translate, it is necessary to repeat the activity on "many" projects without replacing the generator (ref. EN 50128, B.65 [Ref 2]). The code generated for a single application, however, is valid.
In order to perform the validation process of the translator is thus required to have:

- The model (Stateflow Chart) on which to perform the tests. The chart must be well-formed (i.e., designed according to the requirements and the modeling guidelines).

- An interface model that refers to the original model. The interface model shall be composed of a part of input processing, the reference block and a

part of outputs processing, as represented in Fig. 3.5.

- A set of test vectors (defined as MATLAB arrays) that will be inputs for the interface model and processed by the referred model. The tests should be designed for:
  - full coverage of the requirements;
  - full coverage of the model: the metric for coverage (e.g., decisions, conditions) will be evaluated separately for each project depending on the complexity of the project itself.

  The definition of the test suite is a manual activity: the tests are originated starting from the models and the unit requirements defined during the design phase of the system development, according to the requirement coverage criterion. Automatic functional test generation techniques were not used due to the fact that they generally require models with a higher level of abstraction than the ones we use to generate code. The time needed to devise the tests, by a domain expert, is of the same order of magnitude than the time needed to model the system.

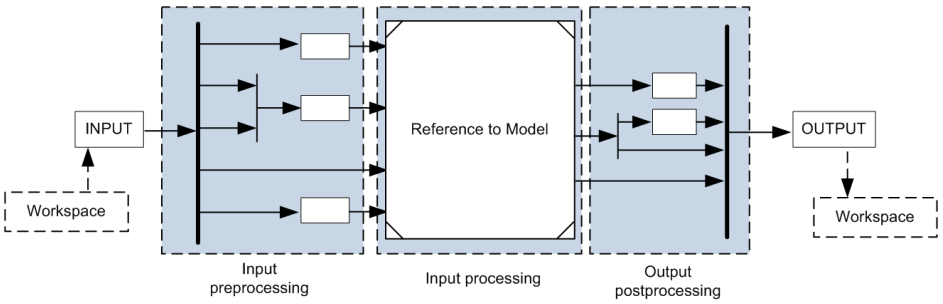


Figure 3.5: Reference model for testing through 2M-TVF

The 2M-TVF aims to provide functionality to validate the process of code generation and code itself. Once the interface model and Stateflow model that represents the system have been provided, the Framework proceeds according to the following steps:

1. Create a reference model from which to generate the code.

2. Generate the code from the reference model.
3. Compile and instrumentat the code for the calculation of the coverage.
4. Create a validation model composed by interface model and generated and compiled code.
5. Run tests on validation model and.
6. Generate a report containing the results of the tests on the model.

It is important to note that:

- The syntax errors are not subject to evaluation by the 2M-TVF Framework, despite the dynamic behavior of the validation process can report them if they are present. The syntax errors anyway shall be taken care of in an earlier phase of the development.
- The model that represents the system is never changed during the whole process, but always used by means of a reference model; the reference model is a special Simulink/Stateflow model that incapsulates the original model allowing to perform operations on the inputs and the outputs of the original model without changing it.

The 2M-TVF execution steps are better detailed in the following.

Reference model creation: the reference model is created with the purpose of generating code both from the reference model itself than from the Stateflow Chart to which the reference model refers. The reference model is a "copy" of the model used for the testing from which have been eliminated the structures for the input pre-processing and ouptut post-processing.

Code generation from the reference model: the code is generated from the reference model by means of RTW Embedded Coder [80]. The generator configuration must be the same used every time it generates the code to be compiled and installed on the target.

Code instrumentation and compiling: the code is compiled and instrumented in such a way as to obtain information on the coverage once it is executed. The choice of the compiler and the linker has fallen on gcc v3.4.3 [31], which holds the proven-in-use property and provides also built-in functionalities for code coverage measurements.

Validation model creation: the validation model is created starting from the reference model, which is copied and enriched with additional features, such as a Simulink block that allows to integrate and run binary code into the Simulink environment; to the "binary code block" are given the same inputs that are passed to the reference model (and thus to the model itself). The outpus of the "binary code block" will be processed exactly as the ones of the reference model.
The structure of the model validation is shown schematically in Fig. 3.6.

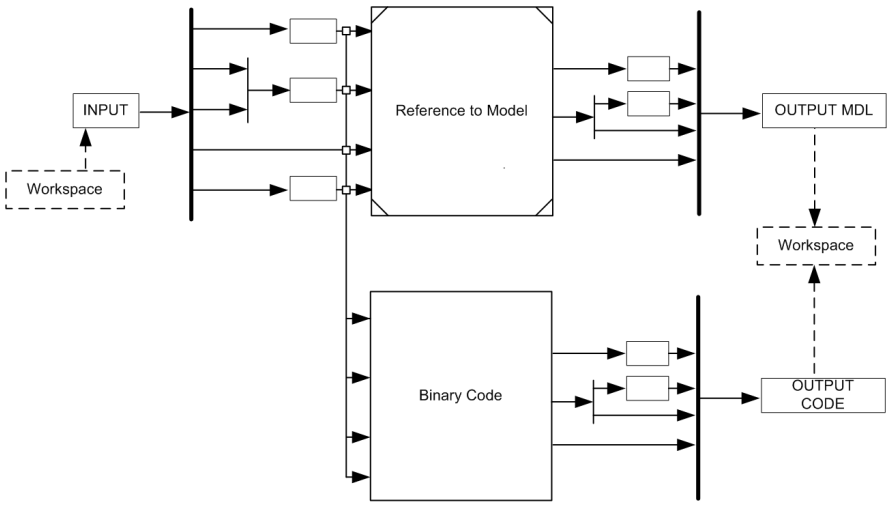


Figure 3.6: Structure of the validation model

Tests on validation model: the tests are defined as matrices in MATLAB and designed to achieve the requirements coverage and full coverage of the model according to the metric choice, are stored on the basis of MATLAB

Workspace and from there sent to the model during the validation phase of simulation. The data is then taken from the Workspace and passed in input to the model of the system (via the reference) and code. In this way both the high-level representation of the system and the generated code are exercised.
At the end of the simulation results are stored in the Workspace and compared. In case of discrepancies it will be necessary to justify the different behavior between the model and run the code.

Report generation: after execution of the tests it will be automatically generated a report containing information about the validation model, the simulation results and coverage .

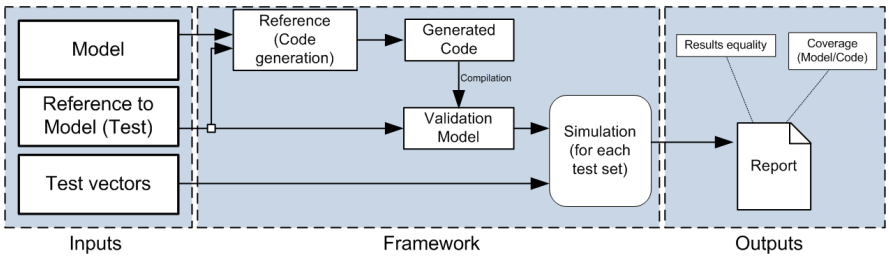


Figure 3.7: 2M-TVF structure

The major advantage of the adoption of 2M-TVF is that the whole verification process is conducted internally to the Simulink environment, and it is completely automated.

3.2.2 Formal verification of model properties

The CENELEC EN 50128 norm states that formal verification techniques, such as model checking [14] and theorem proving [13], are highly-recommended practices for safety-related software. Despite the norm guidelines and the successful history of formal verification in academia, the adoption of these technologies by companies is still more the exception than the rule. For example, in 2006, Honeywell started defining an approach for the translation of Simulink models into the input language of the SMV model checker [45], or the Rockwell Collins

methodology [47], that starts from Simulink/State ow models to derive a representation into the Lustre formal language, and then perform code generation (in C and ADA) and formal verification through different engines such as NuSMV and Prover.
Formal verification tools are still perceived as too much complex, requiring highly specialized personnel that is able to deal with formal languages and temporal logic. Furthermore, there is still no clear evidence on the actual benefits in terms of cost reduction given by this technology with respect to traditional testing.
With the Metrô Rio project, GETS decided to perform a systematic experimentation with formal verification by means of Simulink Design Verifier (DV), a test generation and property proving engine based on Prover Technology [1], which uses a proprietary algorithm based on Bounded Model Checking (BMC) with SAT-solvers [16] combined with K-induction [72] techniques. It was not the first time that the company practiced formal verification. In previous projects, the two open source model checkers SPIN and NuSMV were evaluated for the verification of railway interlocking equipments [25], and Statemate ModelChecker was used in the development of a system for object detection in level crossing areas [6]. In this case the choice fell upon DV, since the tool uses Simulink/Stateflow as modeling languages and it is fully integrated in the Matlab environment. These aspects have been considered as favorable elements to disrupt the reluctance of the developers towards formal verification.

	Requirement
1	*If an information point with authorized speed equals to zero is received, the system shall raise the Train Trip (TT) event*
2	*If the TT event is raised, the TT procedure shall be activated*
3	*If the TT procedure is active, it shall remain active until the train is not standing*
4	*If the TT procedure is active, the brake shall be activated*
5	*If the TT procedure is active and ICO_TT is invisible, ICO_TT shall start blinking*

Table 3.1: Extract of unit requirements for Metrô Rio project

The focus is again on units, and verification is performed at the level of the Stateflow models. The main objective was to understand if, with the new formal development process, the formal verification task could replace the costly unit testing task.

Verification through DV is performed by translating the property that one wishes to verify into a formula expressed in the Simulink language. In the case of Metrô Rio, the properties are the unit requirements we obtained through the system functional requirements decomposition. The formula derived from the property has the form of a graphical circuit where the variables considered by the property are connected by Simulink blocks implementing logical (AND, OR, NOT, etc.), arithmetic ($+$, $-$, $\times$, etc.) and time delay operators. The engine verifies that the formula is globally true for every execution path of the Simulink/Stateflow model. The property is interpreted as if both the A and G operators of the popular CTL (Computation Tree Logic) paradigm would be prefixed to it: it is basically an invariant. If the property is violated, a counterexample showing a failing execution is given in the form of a test case for the model.
As an example of property representation, consider the fifth unit requirement of Table 3.1, related to the Train Trip function[4] of the system. This requirement can be represented in the form of a Simulink circuit as depicted in Fig. 3.8. Following

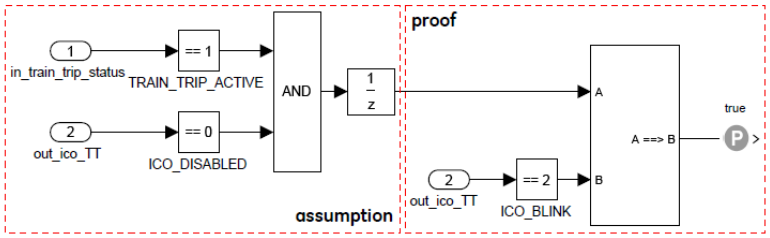


Figure 3.8: Fifth requirement of Table 3.1 represented in the form of a Simulink circuit

the terminology of DV, the first part of the formula represents the *assumption*, which is a set of hypotheses on the state of the model (i.e., the value assumed by the input or output variables of the module). The second part of the formula represents the *proof*, which is a set of expected values on the output that one wants to hold whenever the hypotheses are met. The time delay operator of the assumption is used to ensure that the proof is checked one execution step after the verification of the assumption. The requirement has an equivalent CTL representation: $AG((in\_train\_trip\_status = 1 \wedge out\_ico\_TT = 0) \rightarrow AX(out\_ico\_TT = 2))$. Such an equivalent CTL representation is not easy to

[4] Train Trip is the normally the function concerning the control over the unauthorized passing of a red signal

give for more complex requirements, which can however be represented in the *Assumption/Proof* form. Consider again the Train Trip function: once the train comes to a standstill after the emergency brake issued by the system, the Train Trip icon is supposed to stop blinking. Then, the driver pushes the icon for at least one second, the icon disappears, and the system releases the brake so that the train can leave again. We evaluate the following sub-requirement involved in this function: *If the Train Trip icon (ICO_TT) is visible and not blinking, the train is standing and ICO_TT is pressed for at least one second, ICO_TT shall become invisible*. The corresponding Simulink circuit is represented in Fig. 3.9. In this requirement a timer variable is involved: the button associated to the icon shall remain pressed for at least one second before ICO_TT can change its state. In the assumption part of the formula, two subsystems are used to evaluate the duration of pressure. This is supposed to be more than 84 main cycles (each main cycle has a minimum duration of $12ms$, therefore $1000ms/12ms \approx 84$) to enable the proof part of the formula. The CTL representation of this property is in principle feasible, but requires the usage of 84 nested sub-formulae, each using the next operator.

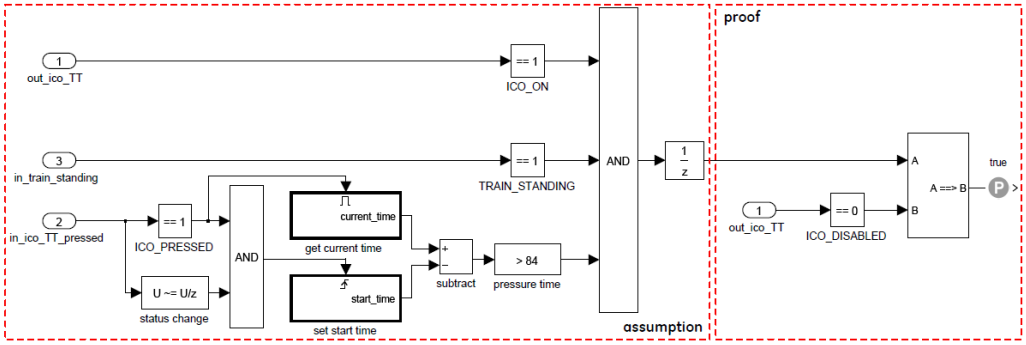


Figure 3.9: Unit requirement with timer in the form of a Simulink circuit

As one can infer from Table 3.1 and from the examples, all the unit requirements used in this context have the form *if* $< \Phi >, < \Psi >$. Φ is a condition sentence that might involve entities related to input and output variables in the model, while Ψ is an action sentence that involves entities related to output variables only. The translation of these requirements into the *Assumption/Proof* form is rather natural, and, in our requirements set, we have identified three classes of requirements, according to the structure of the formula resulting from the

translation:

- **class 1** requirements where Φ involves input variables only. These can be translated into a formula $assumption(I)/proof(O)$, where I and O are (possibly empty) subsets respectively of the input and output variable sets of the Stateflow model to be verified. This class of requirements includes requirements 1, 2 and 4 of Table 3.1. Invariants that are independent from the input also belong to this class.

- **class 2** requirements where Φ involves both input and output variables. These can be translated into a formula $assumption(I, O_i)/proof(O_j)$ where O_i and O_j are (possibly disjoint) sub-sets of the output set. This class includes requirements 3 and 5 of Table 3.1.

- **class 3** requirements having any of the previous forms, but where the Φ or Ψ require some value to hold for a certain amount of time. These can be translated into a formula $assumption(*, T_i)/proof(*, T_j)$, where T_i is a set of timers on the variables of Φ, and T_j is a set of timers on the variables of Ψ. The requirement associated to the circuit of Fig. 3.9 belongs to this class.

Requirements that do not belong to any of the classes are not verifiable with DV. Among these requirements we consider both non functional requirements, normally verified through model inspection also when model-based testing is applied, and requirements that involve adherence of the output to complex mathematical functions, such as the ones related to the computation of the braking curve (a *braking curve* is used to calculate how long it will take a train to stop from a given speed). In this case, the verification approach adopted has been model duplication: a continuous model of the curve has been created, and its behavior has been simulated and compared to the discrete model used for code generation. We have translated and verified 80 out of the 377 verifiable unit requirements. Our experiments have shown that requirements belonging to the first class need 10 minutes of set-up time on average, while for the second class we reach 15 minutes. The verification time is in both cases negligible (less than one minute). The third class requires longer set-up time, around 40 minutes for each requirement, mostly due to the complexity of expressing relationships between events in

the timeline through memory blocks or time delay operators. In many cases the formal verification incurs state space explosion problems and fails to terminate, hence these requirements are likely to need to be verified through testing.
Table 3.2 reports the number of requirements belonging to each class for the Metrô Rio project. It can be noticed that most requirements belong to the first two classes, which required limited set-up and verification time. Requirements belonging to the third class are limited and are mainly related to human-machine interaction (button pressure, icon display timers, etc.).

	class 1	**class 2**	**class 3**	**n.v.**
# requirements	285	73	19	113

Table 3.2: Classification of requirements according to the associated formula

According to these results, a new verification process based on formal verification can be foreseen. Requirements belonging to the first two classes can be formally verified with a cost that is 50% to 66% lower than the one required for testing (each test needs 30 minutes in average). Requirements of the third class and non verifiable cases can be treated with the previously used approaches: model-based testing or inspection. Unfortunately, except for the time elapsed in the verification and the results of the tests, DV does not give any further information about the process, and thus the user is unable to estimate which is the effective cost of the verification, for example in terms of memory consumption.
The main threat to validity of our evaluation is given by the fact that the models were already verified by means of testing: no further error was discovered during the formal verification phase. Our estimates do not consider the iterative task of adjusting models or formulae whenever a counterexample is issued. Nevertheless, we believe that the cost of these adjustments can be compared to the cost of fixing a bug detected by testing.
Introduction of formal verification in the established development process is still at the evaluation stage. Despite the encouraging results in terms of costs, there are other issues that have to be addressed, such as the qualification of the tools and the integration of the approach with the other process tasks. The company is currently defining solutions in these directions.

3.3 MBD and safety-critical systems development: a case study

In this section is detailed an example of development with MBD and formal verification performed during the experimentation in order to be evaluated by GETS, starting from natural language requirements provided by RFI (*Rete Ferroviaria Italiana*) in the context of SCMT project [62].
SCMT is the system of Automatic Train Protection (ATP), which was adopted by the Italian railways over most of the territory, which lets you control the running of the train basing on the aspect of the railway signals, to slow down and respect speed limits imposed for given rail sections. The information submitted by the Trackside sub-system (Sotto Sistema di Terra, SST) are received and processed by the Board subsystem (Sotto Sistema di Bordo, SSB).
The function Presenza Personale di Macchina (PdM) [63] checks the behavior of the drivers and starts emergency procedures if not-safe conditions occurs.
The function is modeled as a finite state machine according to the specifications of the system. Is then tested and verified in a formal way through the tool Simulink/Stateflow to verify that the requirements expressed by the functional specifications [62] [63] are indeed satisfied.

3.3.1 Model requirements and implementation

The system SCMT is composed by a Trackside sub-system (SST) and a Board subsystem (SSB). The SST transmits to the SSB information about the current speed of the train and current signal status. The SSB verifies that the train travel in safety, otherwise it notifies to the driver any violation of a speed limit or active automatically a the braking procedure. In the case of the function Presenza PdM, the purpose is to verify that the driver is behaving correctly from a safety point of view, checking its physical status through acoustic interfaces (sound signals) and mechanical interfaces (buttons). The acoustic signals are activated due to the loss of stationary condition (train is moving without permission) or in response to violations of speed limits. After the acoustic signal has been activated, if there is no action on the mechanical interfaces within a given

amount of time, the system requires the emergency braking.
RFI provided requirements to the GETS company in order to implement the desired functionalities.
The implementation of the module *Presenza PdM* has been made using Simulink/Stateow starting from the given requirements. Each requirement can be part of one of the following categories:

- **[E]**, Essenziale: a prerequisite for the proper functioning of system.
- **[D]**, Differito: essential requirement but delayed in time.
- **[F]**, Facoltativo: no mandatory requirement.
- **[U]**, Instabile: requirement that needs further study or more accurate details.

The requirements specified for the module *Presenza PdM* belong to the [E] type, and then they are essential in the implementation.
The functional requirements regarding Module *Presenza PdM* are divided into requirements capture inputs and requirements management.

Input requirements

The input requirements are summarized in the state diagram in Figure 3.10. The requirements describe the interaction between the module *Presenza PdM* and the other modules composing the SSB.

A subset of the requirements is listed below.

1. The function shall acquire periodically from the Odometry the train stop condition (*Treno fermo, TF*) in order to compute its functioning status.
2. The function shall acquire periodically from the Odometry the current train speed (v) in order to evaluate the activation of the emergency braking.
3. The function shall acquire periodically from the Odometry the space (s) in order to compute when to require the actions of the drivers (*PdM*).
4. The function shall acquire periodically from the Clock the current time (*RTC*) in order to compute when to require the actions of the drivers (*PdM*).

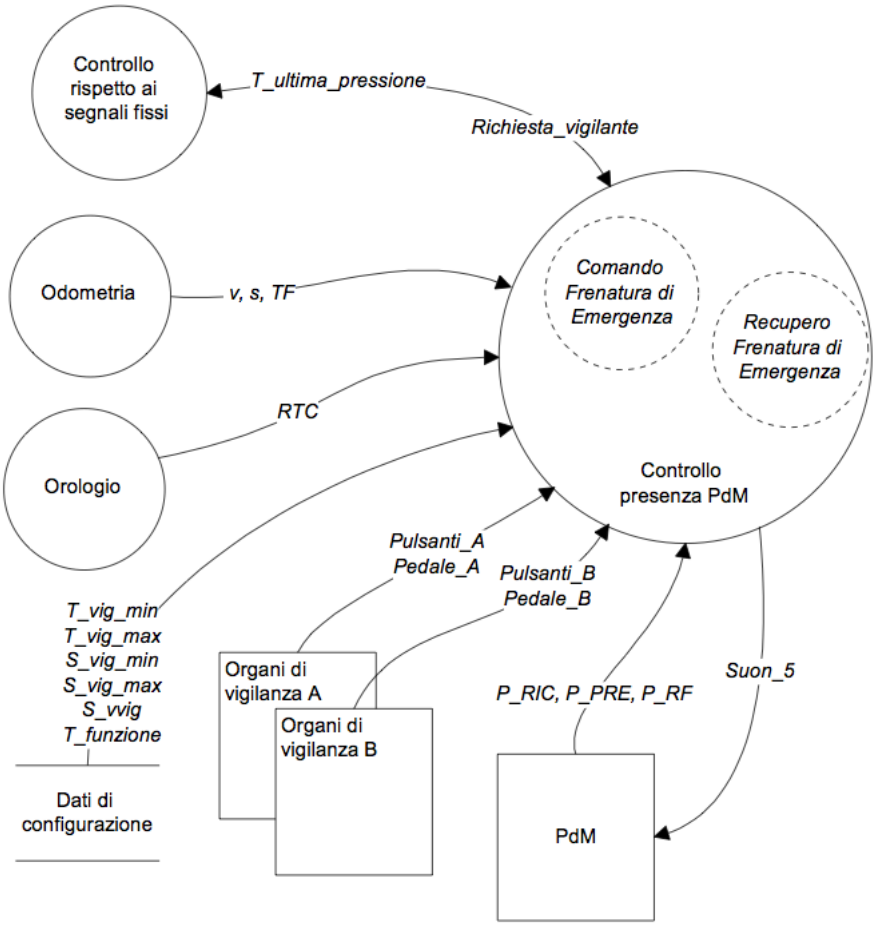


Figure 3.10: Function context diagram

5. The function shall acquire periodically from mechanical interfaces (*Pulsanti_X, Pedale_X* with $X = A$ *o* B according to which Cab is activated), the intervention of mechanical interfaces by the driver in order to check the driver's activity.

6. The function shall acquire from the Configuration Data the speed threshold (*S_vvig*) for deciding the changing of control of driver actions from time-based algorithms to speed-based algorithms (first hypotesis value for the threshold is 100 Km/h).

7. The function shall acquire from the Configuration Data the time *T_vig_min* to set, when the train has speed $v > S\_vvig$, the driver's intervention

minimum time threshold for button pressing (first hypotesis value for the threshold is 2.5s).

8. The function shall acquire from the Configuration Data the time T_vig_max to set, when the train has speed $v > S\_vvig$, the driver's intervention maximum time threshold for button releasing (first hypotesis value for the threshold is 30s).

9. The function shall acquire from the Configuration Data the space S_vig_min to set, when the train has speed $v <= S\_vvig$, the driver's intervention minimum space threshold for button pressing (first hypotesis value for the threshold is 70m).

10. The function shall acquire from the Configuration Data the space S_vig_max to set, when the train has speed $v <= S\_vvig$, the driver's intervention maximum space threshold for button releasing (first hypotesis value for the threshold is 830m).

11. The function shall acquire periodically from the Emergency Procedure module the information about the current status of emergency braking procedure (active / not active) in order to be able to suspend driver's control activities and re-initialize itself.

Starting from the requirements, the model in figure 3.11 has been designed.

In the model were included inputs, which are received by the module *Presenza PdM* from other modules, and the outputs that the module provides to the other ones. Concerning the configuration data, they represent fixed parameters and therefore were modeled as constants, whose values are those of the first hypotesis specified in the requirements.

It is important to notice that in the model were also added outputs not given by the requirements in order to be able to verify properties through the Simulink Design Verifier tool.

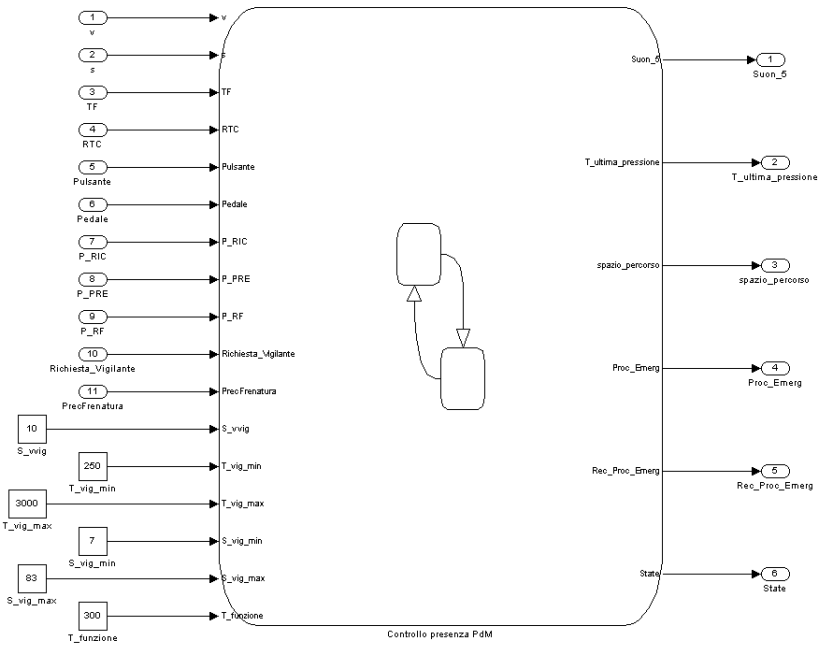


Figure 3.11: Simulink model

Management requirements

The management requirements details how the model shall react basing on the inputs and the internal state. In the case of SCMT the requirements were already in the form of a Finite State Machine (FSM), represented in 3.12.

The Model-Based Design is particularly suitable in case of FSM-like requirements, since it is possible to have a one-to-one corrispondence between the states defined by the requirements FSM and the designed model (depicted in figure 3.13).

In order to better understand the development process, a list of requirements and their implementation are listed below.

Req 1: The function shall behave as the FSM represented in Fig. 3.12.

Req 2: At the activation, the function shall be in state *INIZIALIZAZZIONE*.(Fig. 3.14)

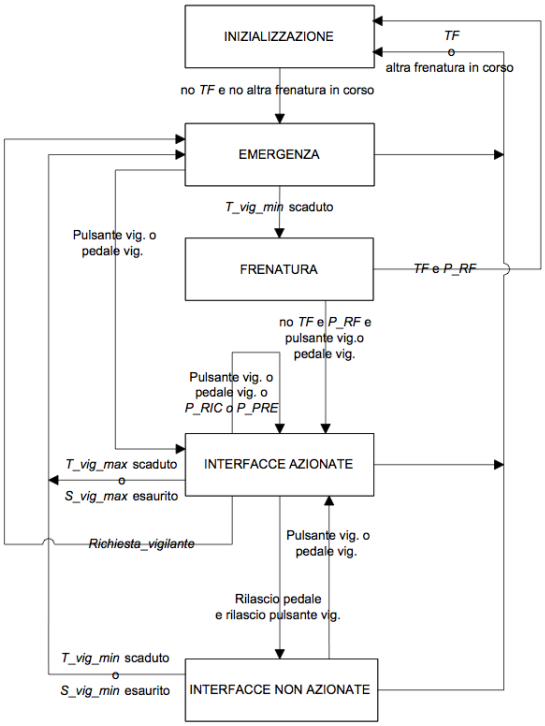


Figure 3.12: Requirements FSM

Req 3: Independently on the current state except for the state *FRENATURA*, the function shall return in the state *INIZIALIZZAZIONE* if it is restored the train stop (TF) condition or if it is currently happening an emergency braking requested by another function(Fig. 3.15).

Req 4: In state *INIZIALIZZAZIONE*, the function shall disable all the currently active timing checks and every acoustic notification (Fig. 3.16).

Req 5: From state *INIZIALIZZAZIONE* the function shall transit to state *EMERGENZA* (Fig. 3.17) if:

- No emergency braking has been requested by other functions.
- The train stop (TF) condition is lost.

Req 6: When the function transits into state *EMERGENZA*, it shall activate a timer equal to T_vig_min and an acoustic notification.

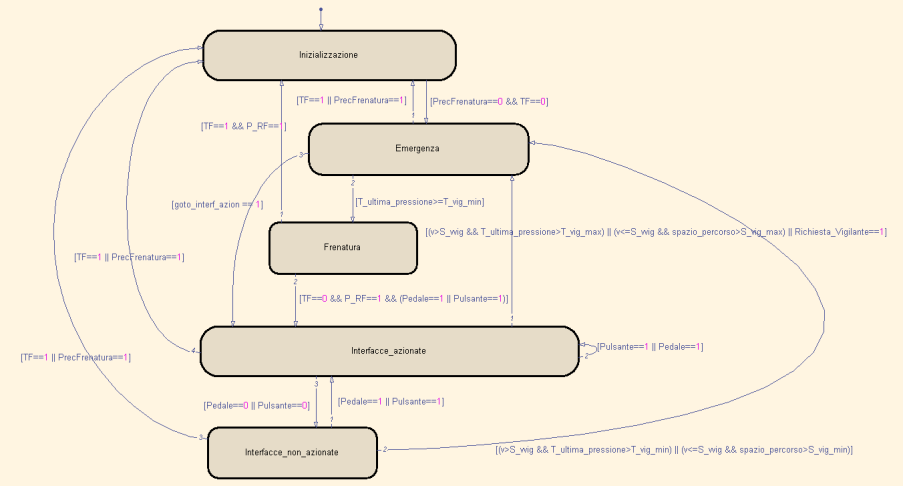


Figure 3.13: PdM Model chart

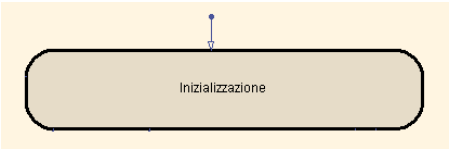


Figure 3.14: Activation of the SSB

It was decided to define three sub-states inside the state EMERGENZA: *Emergenza_init*, *Emergenza_p_premuto* ed *Emergenza_p_rilasciato*. The first one activates the acoustic notification (*Suon_5=1*) and starts computing elapsed time and travelled space, while the other two states are consequence of the Req. 8 and will be detailed later.

Req 7: The function shall transit from state *EMERGENZA* to state *FRENATURA* if the timer *T_vig_min* elapses (Figure 3.19), activating the emergency procedure *Procedura di Emergemza, Proc_Emerg*, deactivating the acousitc notification *Suon_5* and activating the recover emergency procedure (*Procedura di Recupero Emergenza, Rec_Proc_Emerg*) in order to permit the re-enabling of the braking functionalities once the train has stopped (Figure 3.20). Once the *T_vig_min* elapsed (i.e., when *T_ultima_pressione >= T_vig_min*), the system transits from state *EMERGENZA* to state *FRENATURA*(Figure 3.19).

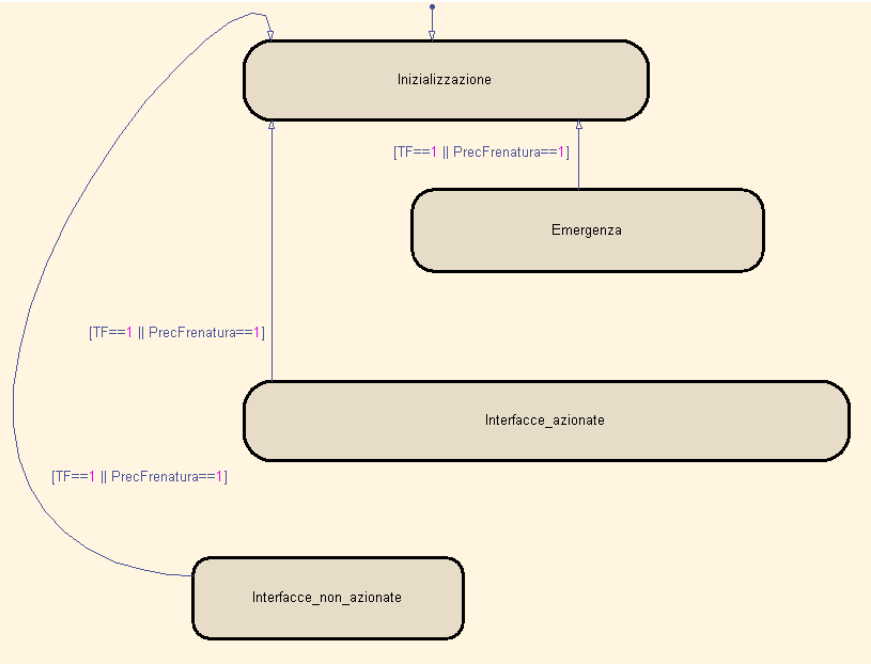


Figure 3.15: Return to state *INIZIALIZZAZIONE*

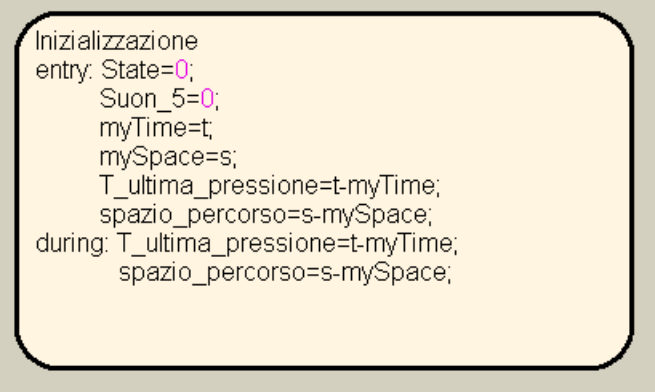


Figure 3.16: State *INIZIALIZZAZIONE*

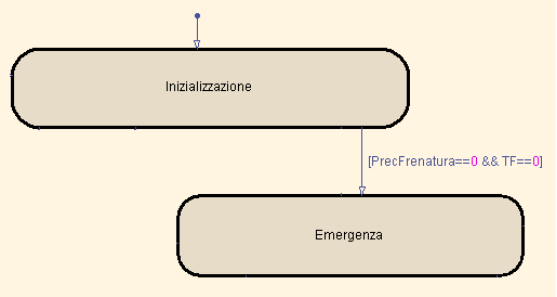


Figure 3.17: From state *INIZIALIZZAZIONE* to state *EMERGENZA*

Req 8: the function shall transit from state *EMERGENZA* to state *INTERFACCE AZIONATE* and disable the acoustic notification if the mechanical interfaces provided to the driver (e.g., button) are pressed and released within *T_funzione* time (Figure 3.21).

In this requirement resides the explanation of the second and third sub-state (*Emergenza_p_pulsante* ed *Emergenza_p_rilasciato*) inside the parent state *EMERGENZA*: in fact once a button is pressed when the system is in the parent state, the second sub-state activates and the check for the pressure time starts. When the button is released, if the pressure time is within *T_funzione*, the system transits into the third sub-state which will make the system evolve into the state *INTERFACCE AZIONATE*.

3.3.2 Model verification

Once the Simulink model and the contained charts have been designed, the properties that the system has to hold can be verified through the integrated tool Simulink Design Verifier [83]. The tools represents an extension of the Simulink tool and permits to verify and validate formally the properties on a model.
The properties to verify are detailed through two blocks in the model:

- *Assumption:* it is placed on the inputs of the model. It determines the domain of interest for the test and it expresses concepts such as "If the input x assumes the value x0..."
- *Proof:* it is placed on the outputs of the model. it determines the expected

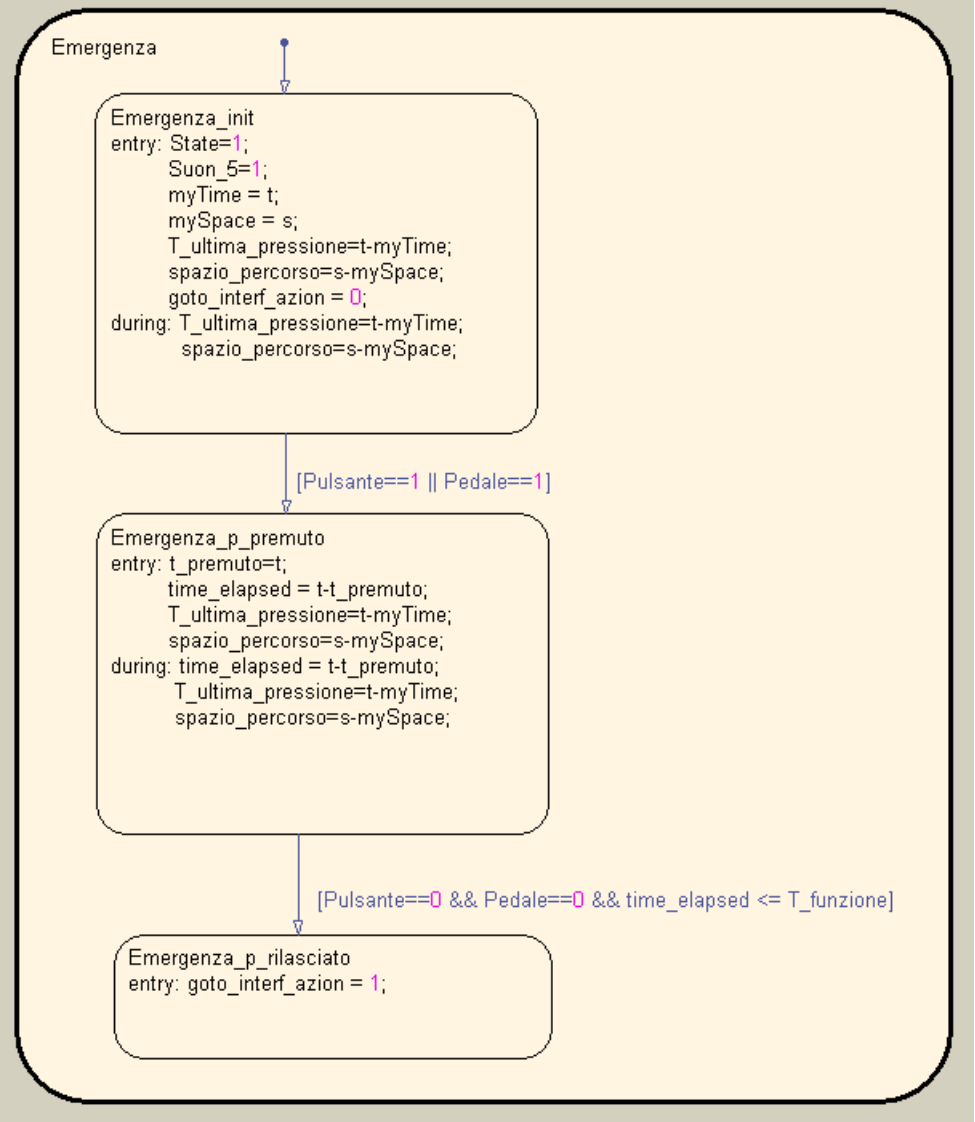


Figure 3.18: State *EMERGENZA*

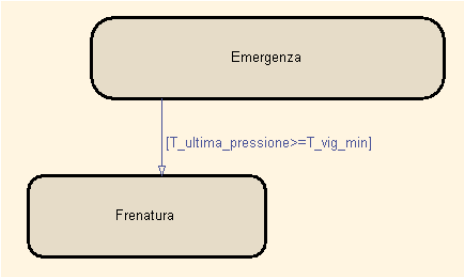


Figure 3.19: From state *EMERGENZA* to state *FRENATURA*

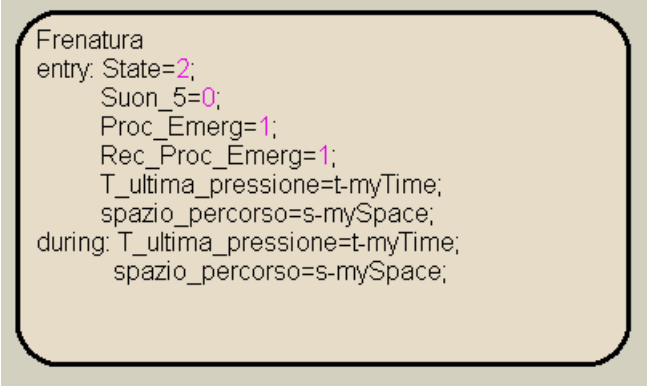


Figure 3.20: State *FRENATURA*

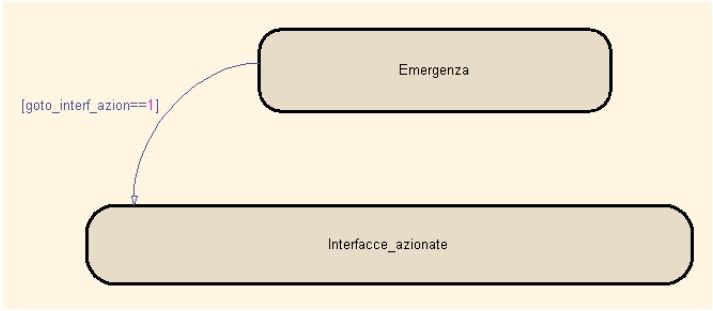


Figure 3.21: From state *EMERGENZA* to state *INTERFACCE AZIONATE*

value for the test. expresses concepts like "...then the output y is y0.".

The tool checks if the property is true in the model and otherwise generates a counterexample. There are also two different strategies for the verification of the test:

- *Prove:* translate the model into a formal language and perform a formal verification on the translated model.
- *Find Violations:* limits the previous technique to a limited number of steps established by the user.

Usually the technique that is preferred to use is the Prove one. In cases where the complexity is too high and the execution of tests fails, the Find Violation technique can be used as a sort of "debug", increasing the confidence on the model behavior, but without any formal proof on the properties of the model.
A simple requirement is representable with a series of blocks Assumption/Proof placed on the arcs of the input/output; in more complex cases must instead ensure that the system is already in a certain state, and this requires further instrumentation of the model.
The complexity in the verification of a requirement is determined by two characteristics:

- The structure of the hypothesis. A requirement may relate to a simple combination input and / or a particular state of the system (or a subset of states). In order to model the hypotesis the adopted technique involves the creation of special blocks that takes as input the current state of the system (or other information useful for the property to prove) and bounds the verification only to the right cases.
- The structure of the thesis. The verification may require the outputs to observe only the value of output (e.g., "the acoustic notification is 'on"') or to observe a change in the value of the output (e.g., "the machine goes from state A to state B") .

A subset of the management requirements that have been verified through *Prove properties* functionality of Simulink Design Verifier is listed below. Note

that, in order to speed up the verification process, the timing constraints have been scaled by the same factor (e.g., in the case of the verification of *Presenza PdM*, the timing constants have been scaled by a factor 10). Requirements have been verified both through Assumption/Proof constructs (if requirements were "simple enough") or through a more complex Simulink circuit, when some check are needed to enable the verification or not.

3.3.3 Simple requirements verification

The first requirement is a simple one, which does not need any elaboration in order to be verified. This requirement belongs to the "class 1" (see §3.2.2).

> **Req 1:** At the activation, the function shall start from state *INIZIALIZZAZIONE*

The Simulink model that follows simulate the requirement considering an Assumption block with value 1 on the TF input (train stopped), in order to model the fact that at the activation of the system the train is not moving, and a proof block with value 0 on the exit "State", where 0 is the value for the state *INIZIALIZZAZIONE*.

3.3.4 Complex requirements verification

This requirement is complex enough to require the creation of a Simulink circuit to check the current status of the system before the verification can be actually performed. This requirement falls in the "class 2" requirements (see §3.2.2).

> **Req. 2:** Independently on the current state except for the state *FRENATURA*, the function shall return in the state *INIZIALIZZAZIONE* if it is restored the train stop (TF) condition or if it is currently happening an emergency braking requested by another function.

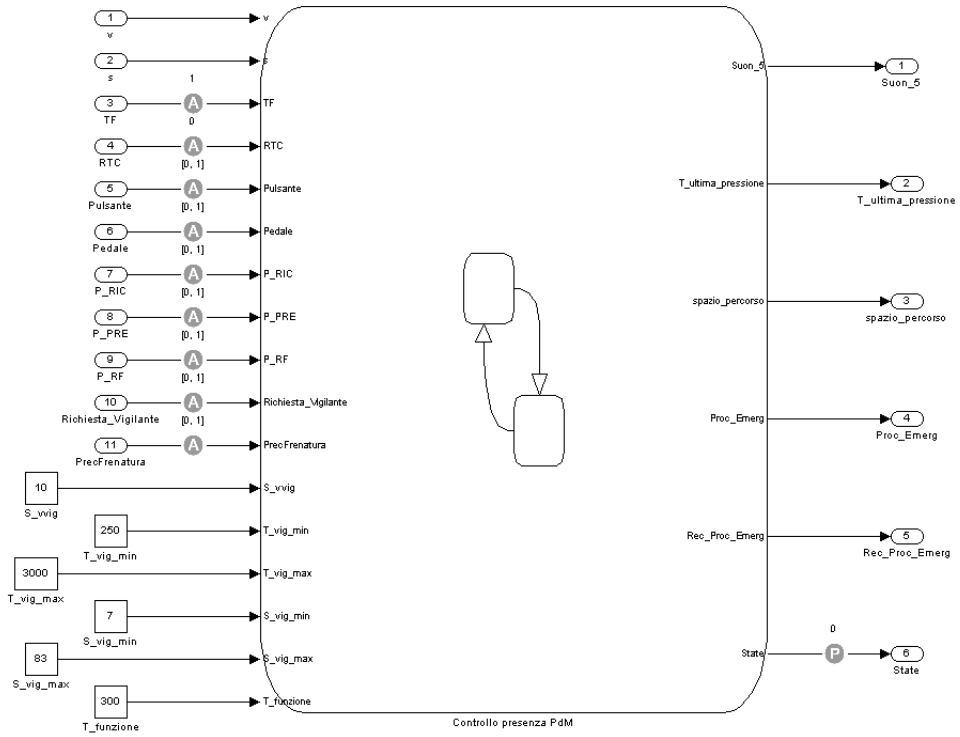


Figure 3.22: Req. 1 verification

The following Simulink model is used to perform the verification.

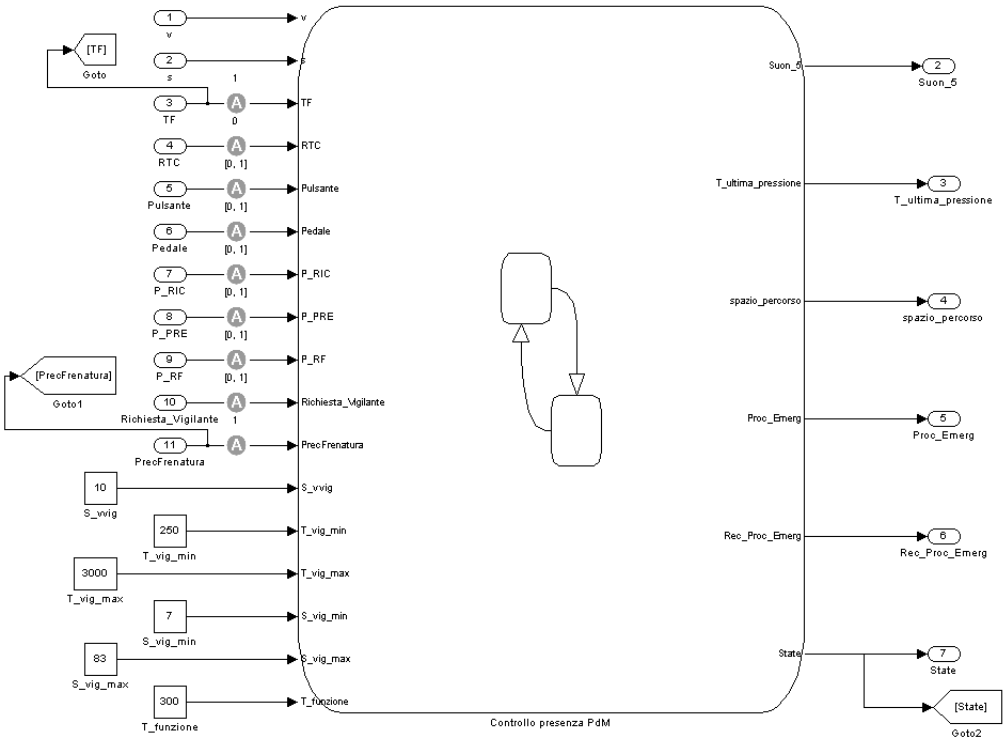


Figure 3.23: Req. 2 verification

According to the requirement, the state *FRENATURA* shall not be involved; furthermore the two inputs on which the Assumption blocks are places shall be bonded through the OR logical operation. For this motivation a Simulink circuit has been designed in order to perform the verification: the EnableProof block implementes the check for the current state (different from *FRENATURA*) and the logical OR between the two inputs that represents the current state of the braking operation and the train stop (TF) condition. In the Proof block is modeled the check for the transition to the state *INIZIALIZZAZIONE*.

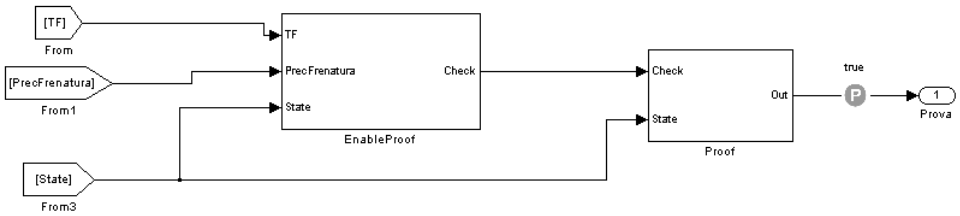


Figure 3.24: Link between EnableProof and Proof blocks for verification of Req. 2

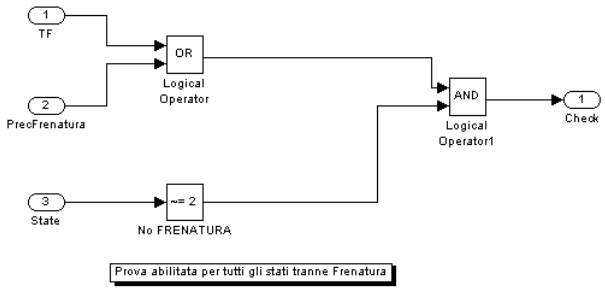


Figure 3.25: EnableProof block for verification of Req. 2

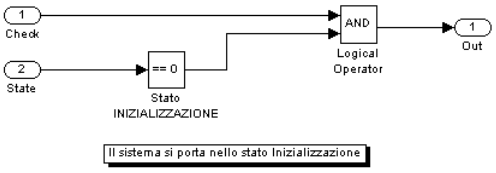


Figure 3.26: Proof block for verification of Req. 2

3.3.5 Timed-related requirements verification

The following requirements are time related, and thus fall in the "class 3" requirements (see §3.2.2). Both verification presented below require the creation of a Simulink circuit to check the current state of the system before the verification can be enabled, and a more complex check that the system remains in that state:

- for at least a certain amount of time (Req. 6);
- within a certain amount of time (Req. 7).

> **Req.6:** the function shall transit from state *EMERGENZA* to state *FRENATURA* if the timers *T_vig_min* elapses, activating the emergency procedure *Procedura di Emergemza, Proc_Emerg*, deactivating the acousitc notification *Suon_5* and activating the recover emergency procedure (*Procedura di Recupero Emergenza, Rec_Proc_Emerg*) in order to permit the re-enabling of the braking functionalities once the train has stopped.

The Simulink model in Fig. 3.27 implements the requirement.

Also for the verification of this requirement the additional two blocks EnableProof and Proof have been used in order to model the conditions for verifications.

In the EnableProof block there is implemented the checks for the current state to be *EMERGENZA* and the check for the elapsed time from the last action of the driver is greater or equal than the requested one *T_vig_min*. Then the Proof block implements the check if the current state has become *FRENATURA*, and the disabling of the acoustic notifications and the enabling of the emergency procedure has occurred according to what stated by the requirement.

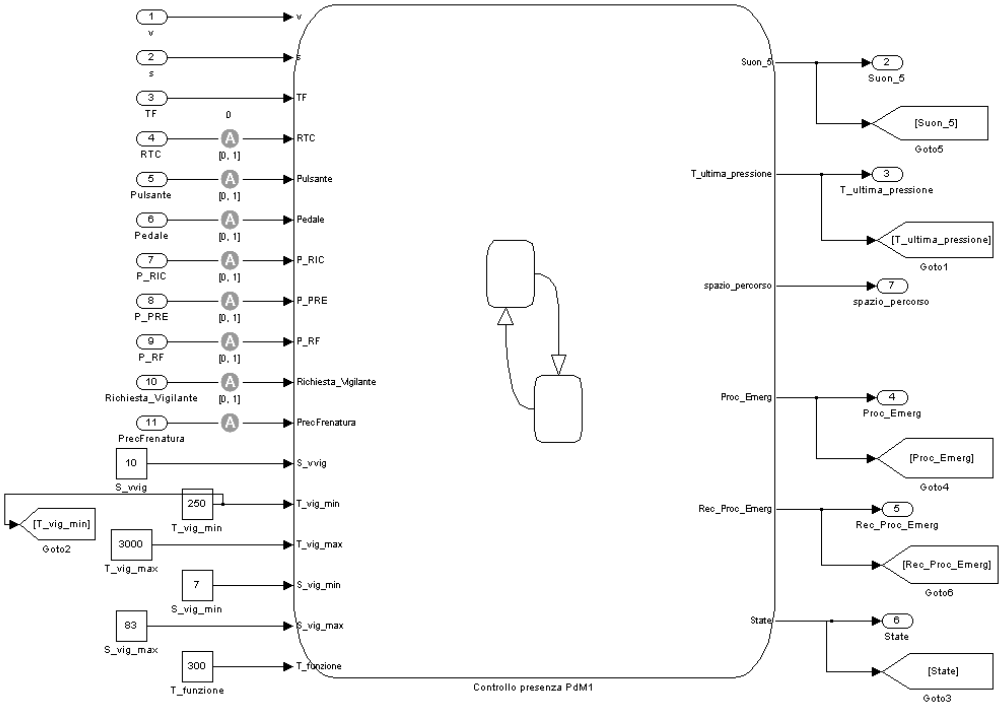


Figure 3.27: Req. 6 verification

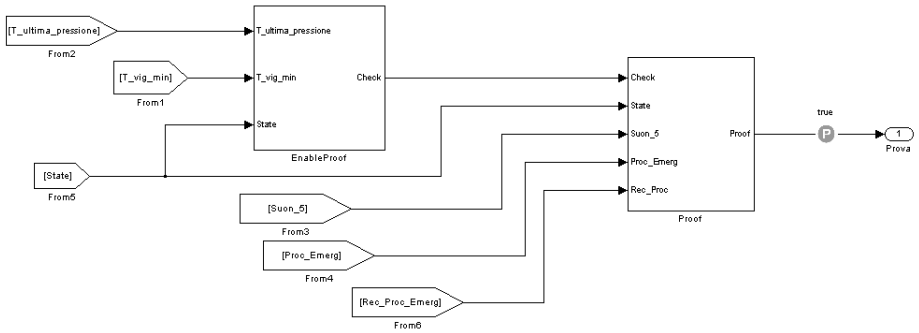


Figure 3.28: Link between EnableProof and Proof blocks for verification of Req. 6

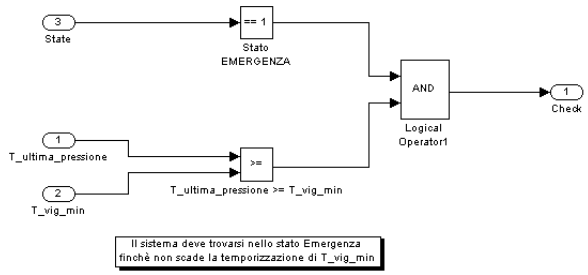


Figure 3.29: EnableProof block for verification of Req. 6

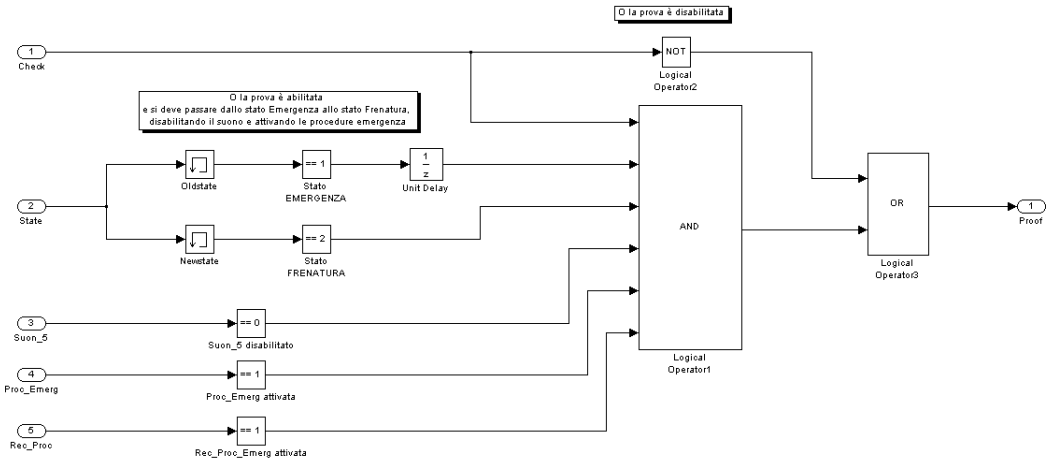


Figure 3.30: Proof block for verification of Req. 6

Req. 7: the function shall transit from state *EMERGENZA* to state *INTERFACCE AZIONATE* and disable the acoustic notification if the mechanical interfaces provided to the driver (e.g., button) are pressed and released within *T_ funzione* time.

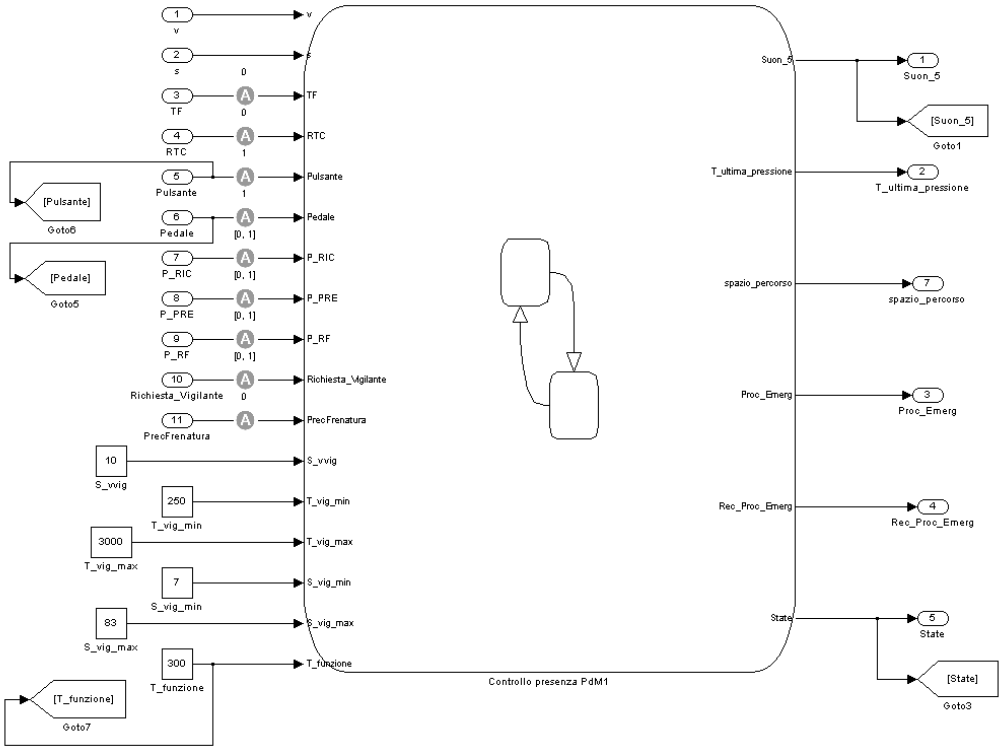


Figure 3.31: Verifica Requisito 7

For the verification of this requirement the block EnableProof (Fig. 3.33) is in charge to check that the button is hold pressed for the right amount of time ([0, *T_ funzione*]); furthermore the block shall also check the current state, that shall be *EMERGENZA*. In the Proof block (Fig. 3.34), if the proof is enabled, it is verified that the transition between state *EMERGENZA* to *INTERFACCE*

AZIONATE and the subsequent disabling of the acoustic notification is performed accordingly to the requirement specification.

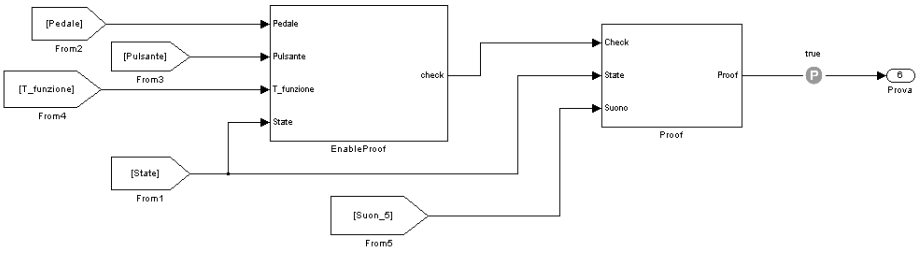


Figure 3.32: Link between EnableProof and Proof blocks for verification of Req. 7

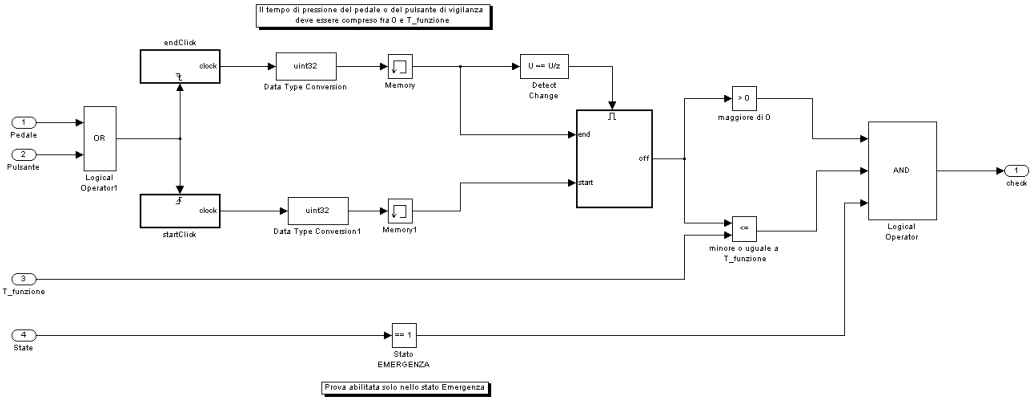


Figure 3.33: EnableProof block for verification of Req. 7

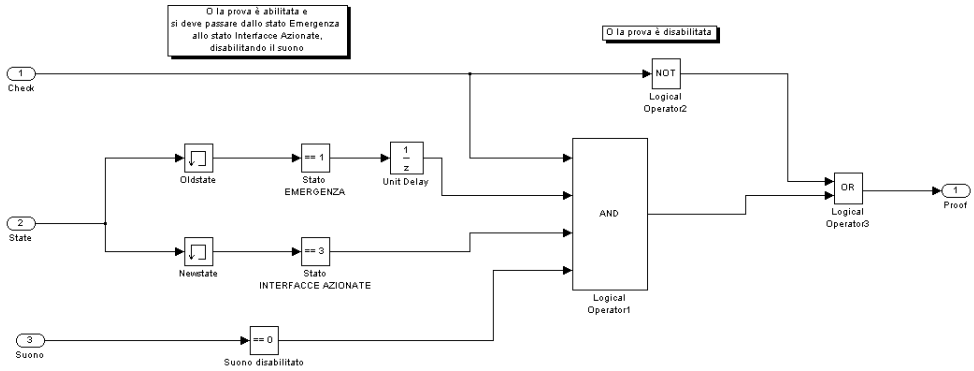


Figure 3.34: Proof block for verification of Req. 7

CHAPTER 4

Formal Methods for Requirements Specification

GETS has recently introduced formal modeling and code generation by means of the Simulink/Stateflow platform, and has defined a model-based process compliant with the CENELEC standards, the set of norms and methods to be used while implementing a railway product for the European market. Simulink/Stateflow are powerful languages for formalizing low-level requirements, while they are less suitable for high-level system requirements specification and analysis.
These activities were normally performed by GETS using a paper-based approach, with natural language documents completed by informal diagrams. Natural language is inherently ambiguous and a more formal means to express requirements was desirable.
So the needs of GETS have moved from the MBD approach, which already became part of the development process, to another level of the development process, the requirement specification.
The company aimed for:

- A formal notation in requirement and architecture specification;
- Tarceability enforcement;
- Cross-phase integrated documentation;

In order to fulfill the company requests, the OMG SysML [55] language was seen as the solution to substitute the traditional text-centric specifications with a formal notation.
This chapter will report the experience of the experimentation of SysML for a small-medium project and the lesson learnt during the experimentation.

4.1 The SysML Language

SysML is a graphical modelling language, developed by the OMG [53] and INCOSE [39], born in response to the UML for Systems Engineering. It is composed by a UML Profile that represents a subset of UML 2 [54] with extensions (Fig. 4.1). It supports the specification, analysis, design, verification, and validation of systems that include hardware, software, data, personnel, procedures, and facilities.
SysML is a visual modeling that provides semantics and notation, but it shall not be confused with a methodology or a tool: SysML is methodology and tool independent.

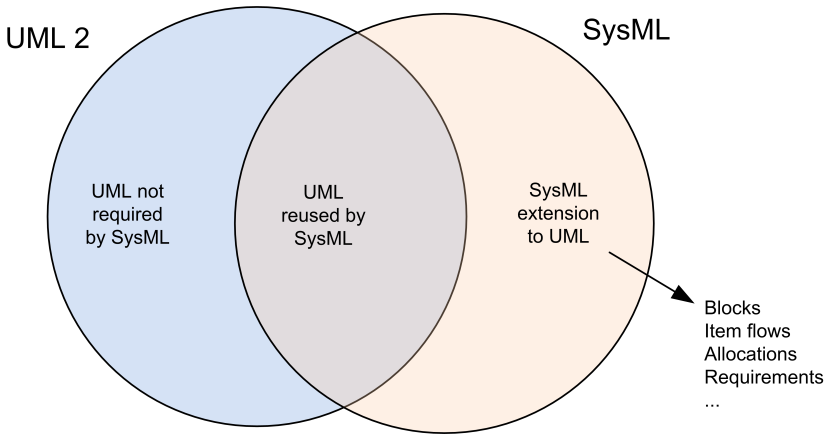


Figure 4.1: Relationship between SysML and UML

The taxonomy of SysML diagrams with respect to the UML2 one is depicted in Fig. 4.2.

Each diagram in SysML represents a model element. The Diagram context is specified in the header, as shown in the Fig. 4.3. The header is thus composed by:

- the diagram kind;
- the model element type (package, block, activity, ..);
- model element name;
- user defined diagram name.

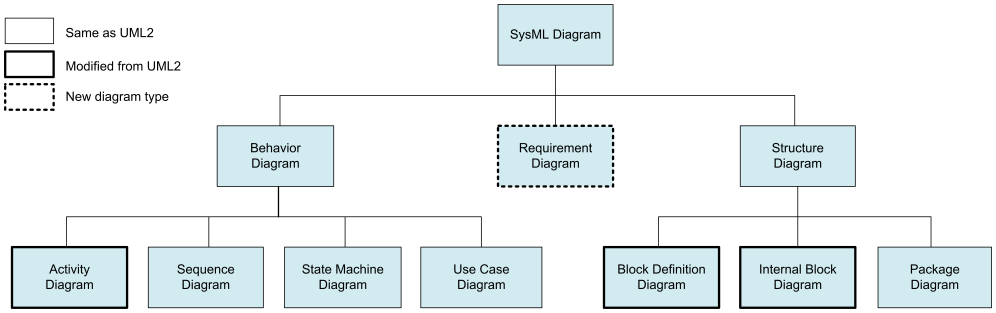


Figure 4.2: SysML Taxonomy

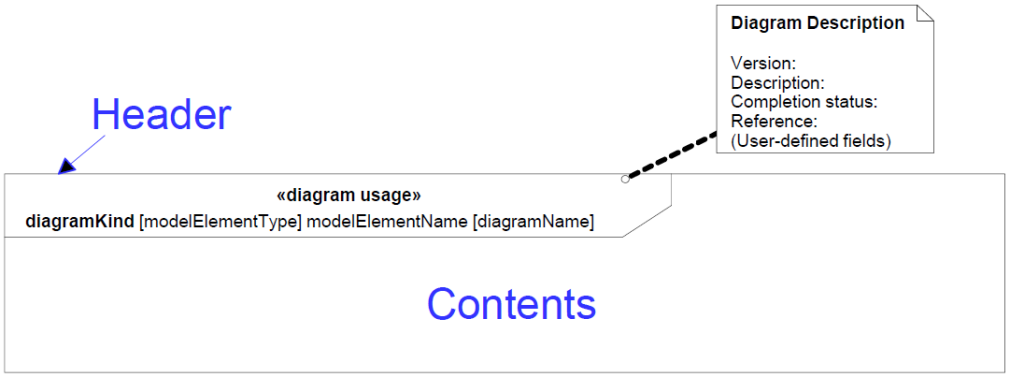


Figure 4.3: SysML diagram header

In the following sections we will detail the basic types of diagram provided by SysML that have been used during the experimentation of the SysML language applied to the development of safety-critical systems in GETS.

4.1.1 Structural diagrams

Structural diagrams are used in SysML to describe the "structure", i.e., the static part of a system that is intended to be modeled. The structure diagrams are not

suitable for describing the data flow between parts, but only which are the parts involved in the system construction.
Structural diagrams are:

- package diagrams;
- block definition diagrams (BDD);
- internal block diagrams (IBD).

Package diagram

A Package diagram is used to organize the model, since it groups the model elements into a name space, allowing to organize the model as a hierarchy or to use viewpoints to augment model organization (see Fig. 4.4). The viewpoints represent the view of a certain user of the system, from the stakeholder to the designer.

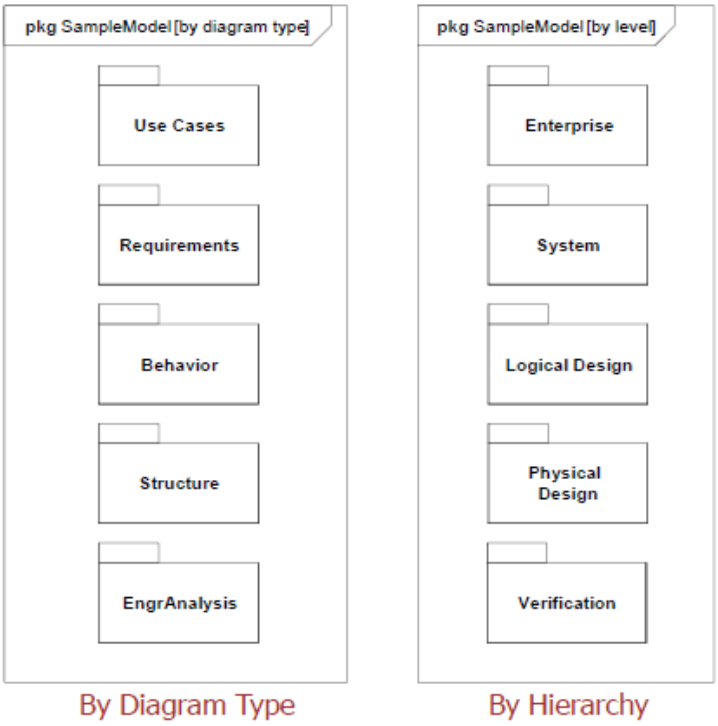


Figure 4.4: Package diagram - model organization

Block definition diagrams and Internal block diagrams

The blocks are basic structural elements based on the concept of UML Class. They provide an unifying concept to describe the structure of an element or a system. The blocks have associated multiple standard compartments that can describe the blocks characteristics (e.g., properties, operation, constraints..) as shown in Fig. 4.5.

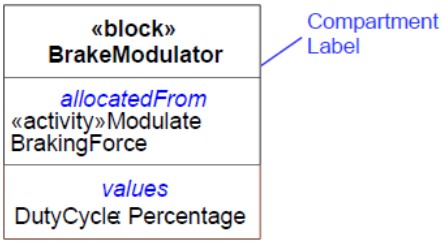


Figure 4.5: Block example

The Block definition diagram (BDD) describes the relationship among blocks, such as composition, association and specialization; differently, the Internal block diagram (IBD) describes the internal structure of a block in terms of its properties and connectors as shown in Fig. 4.6.

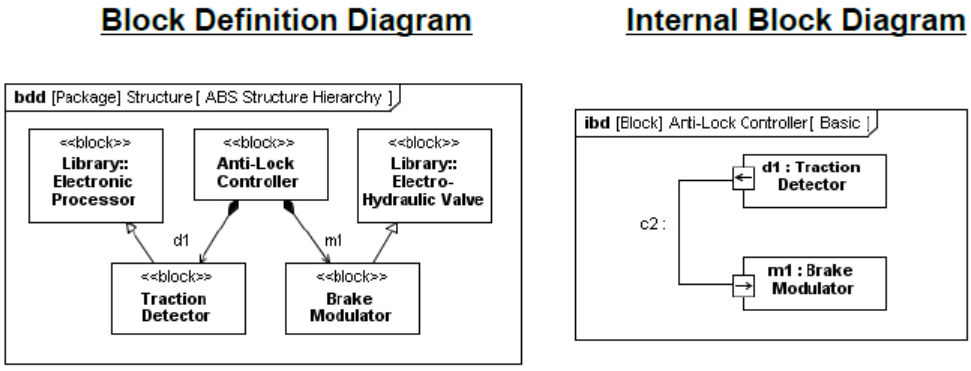


Figure 4.6: BDD vs IBD

The Internal block diagram specifies how the blocks are interconnected, through the usage of connectors and ports as shown in Fig. 4.7. The ports specify interaction points on blocks and parts, and belongs to two classes (see Fig. 4.8):

- **standard UML port:** specifies a set of required or provided operations and/or signals;
- **flow port:** specifies what can flow in or out of block/part.

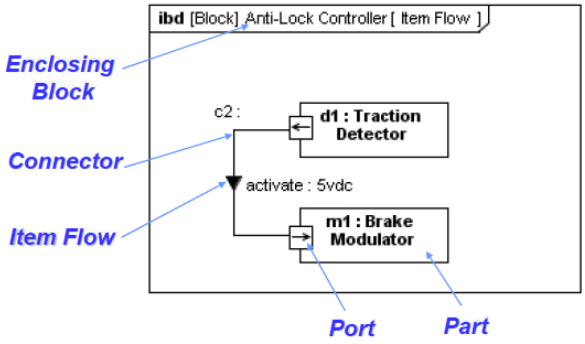


Figure 4.7: Internal block diagram

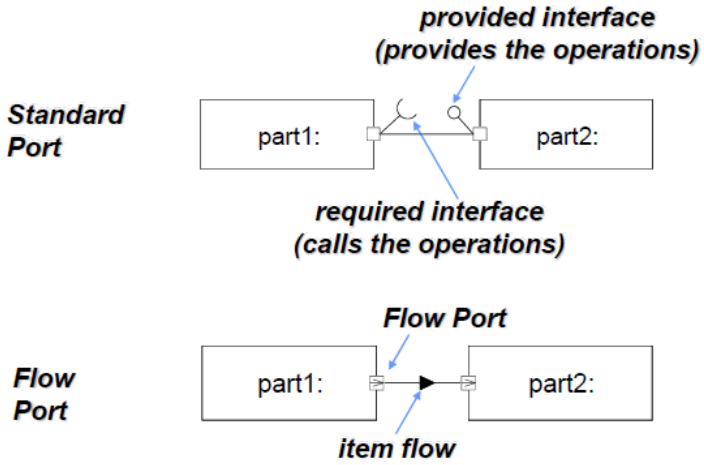


Figure 4.8: Classes of ports

4.1.2 Behavioral diagrams

The Behavioral diagrams detail the behavior of the system, i.e., the dynamic part of the system in terms of interaction between the blocks and the parts of the system.

Among the behavioral diagrams, widely used are the Activity diagrams, which specify the activities, i.e., the transformations of inputs to outputs through controlled sequence of actions (see Fig. 4.9).
The Behavioral diagrams includes also the following UML2 diagrams:

- **use cases diagrams**, which provide means for describing basic functionality of a system in terms of usages/goals of the system by actors;
- **sequence diagrams**, which provide representations of message based behavior, representing flow of control and describing interaction between parts;
- **state machine diagrams**, that are typically used to represent the life cycle of a block, through transition between state (modes of operation) and events (also timed events, unavailable in other types of diagrams).

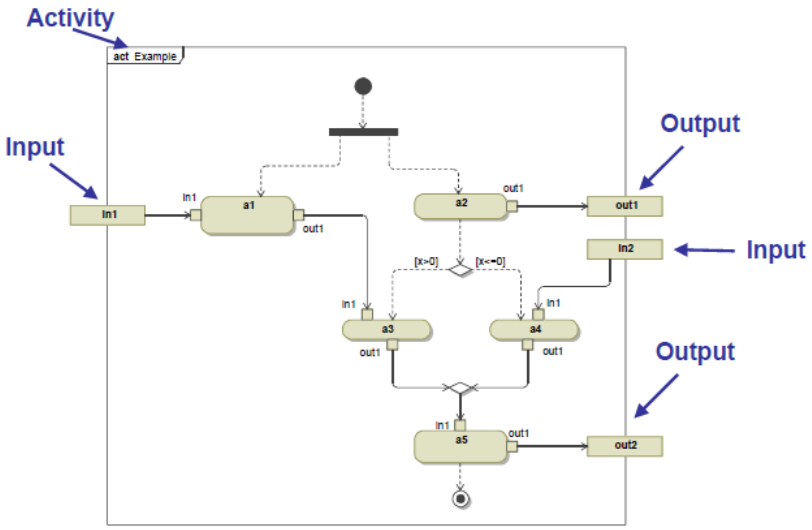


Figure 4.9: Activity diagram

4.1.3 Cross-cutting constructs

The Cross-cutting constructs available in SysML belong to:

- Allocations

- Requirements

The **Allocations** represent general relationship that map one model element to another. Allocations can be of different types, such as behavioral (function is allocated to a component), structural (logical is allocated to physical), software to hardware (see Fig. 4.10), and so on.

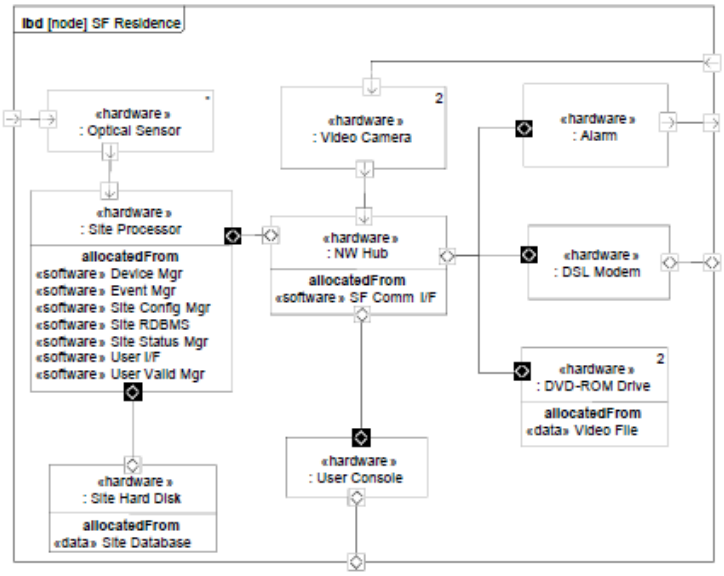


Figure 4.10: Software to Hardware allocation example

The **Requirements** are based on the «requirement» stereotype that represents a text based requirement. The stereotype assign an unique identifier to each requirement among all Requirement diagrams. For each requirement is also possible to add user defined properties such as verification method for that requirement, or add user defined requirements categories (e.g., functional requirement, interface requirement of performance requirement).
SysML defines a visual and graphical representation of textual requirements, specialised associations between themselves or with other elements of the model, and how they can be managed in a structured and hierarchical environment. SysML defines new types of associations (stereotyped dependencies, Fig. 4.11):

- **Derivation** (DeriveReqt) for requirements that are introduced in consequence of another requirement;

- **Satisfiability** (Satisfy) for components (e.g., blocks) that fulfill requirement requests;
- **Verification** (Verify) for diagrams (e.g., activity) that details the verification procedure for a given requirement;
- **Refinement** (Refine) for requirements that refine in greater level of detail the parent requirement;

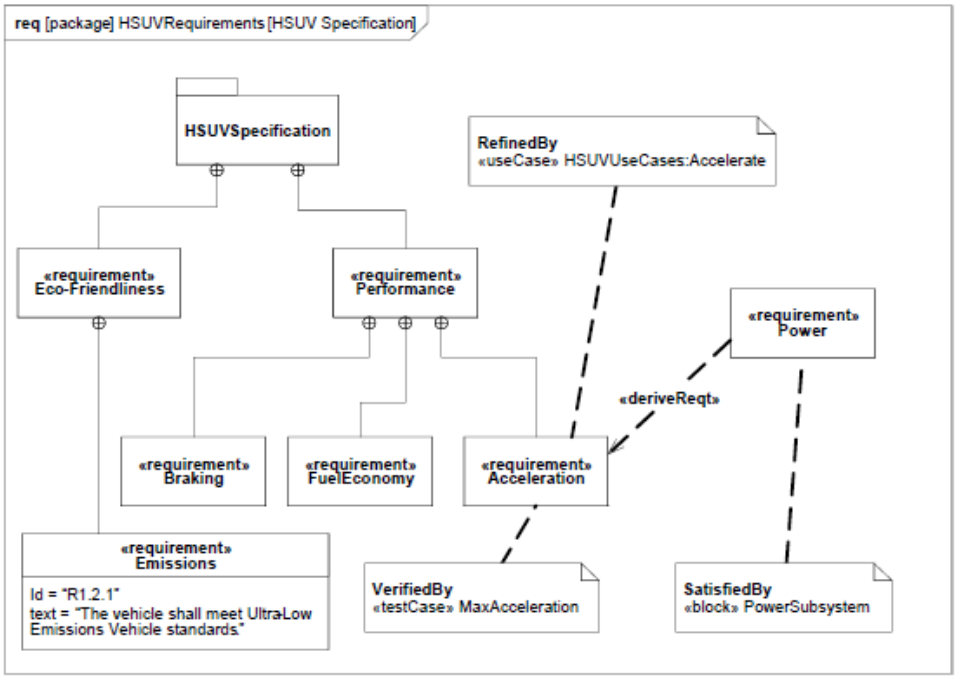


Figure 4.11: Requirements breakdown

4.2 The TOPCASED experience

TOPCASED version 3.0.13 [85] was introduced during the experimentation for the requirements specification of the Failsafe Data Transmission (FDT) system, a platform that manages the switching of adjacent stations. When two stations share a single track for a variety of causes (e.g., the other tracks are busy or not available at the time), the staff of the stations may decide to reverse the direction

of the track from station A to station B, and vice versa (Fig 4.12) to enable trains to pass in both directions, once at time. The reversal of the direction requires that the personnel of the two stations perform a safe procedure; for the whole duration of the procedure they must keep in phone contact. The FDT system allows the exchange of the data necessary to complete this procedure between the two stations, among which is the current occupancy of the track or the current travel direction.

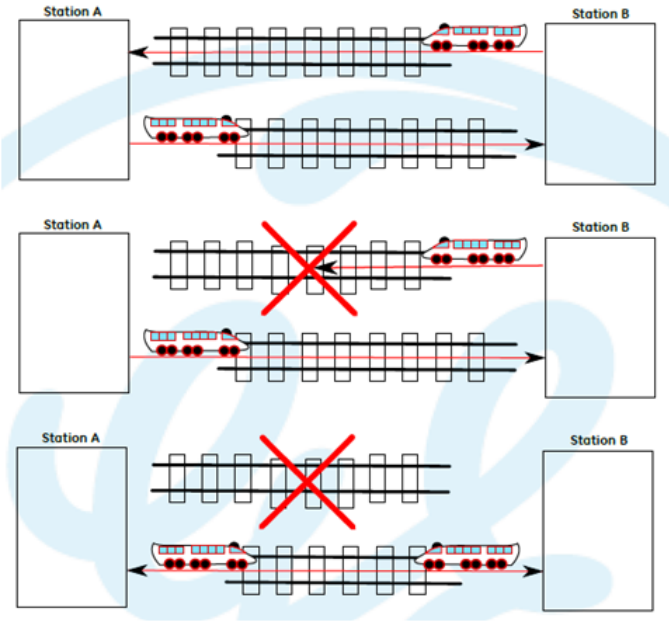


Figure 4.12: FDT System

The project was small enough (6 persons for 7 months) to introduce a new technology, and the tool was perceived as the right candidate to practice SysML, given TOPCASED's claimed orientation to safety-critical systems development. As an example of such orientation, the tool provides a model validation feature, that allows the internal consistency of the produced model and its compliance to the SysML standard to be checked.

A subset of the SysML diagrams was chosen which was considered sufficient to specify the system with a proper degree of detail. This subset was composed by use case, requirement, sequence and structure diagrams (i.e., package, block definition and internal block). The approach planned for the structuring of the different diagrams was aimed at following the CENELEC V-process phases (Fig.

4.13), in order to give a graphical evidence of the adherence to this standard. For this purpose, a single model structured into packages was defined: each package corresponds to a CENELEC phase and includes the diagrams to fulfill the norm prescriptions for that phase. For example, the requirements phase includes mainly use case and requirement diagrams, while the architecture phase is essentially documented with block definition and internal block diagrams. The model was built incrementally, and each artifact of each phase was traced to the elements coming from the previous one.

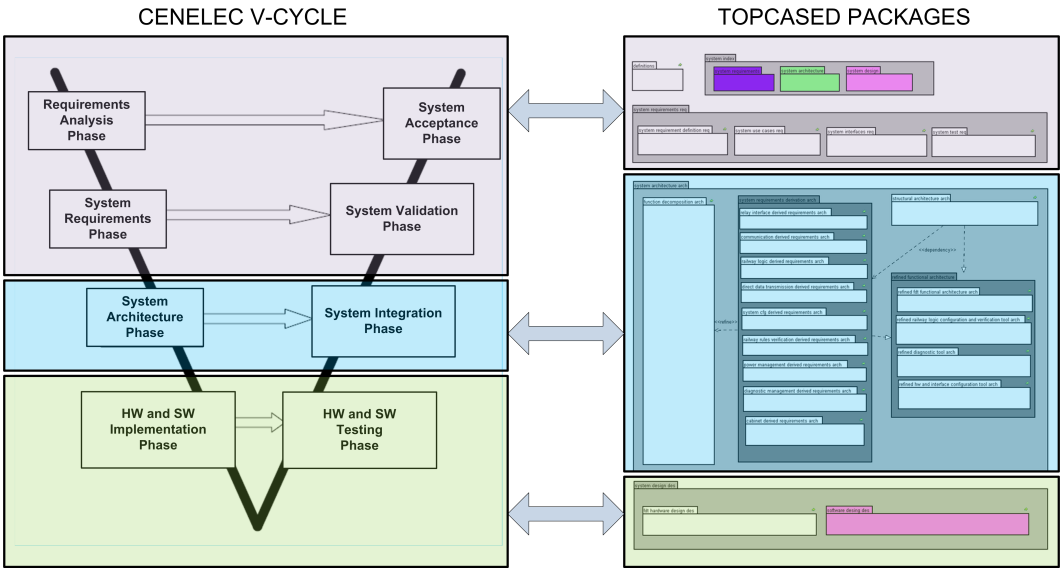


Figure 4.13: TopCased packages for FDT project

The SysML language appeared rather intuitive to users with a UML background, and the tool was easy to learn for people with confidence with the Eclipse platform. In general, electronic/telecommunications engineers encountered more hurdles than software engineers, since some basic principles of the model-view-controller pattern are required for a proficient usage of the technologies. These problems were increased by the absence of a proper documentation for the tool. Despite the large literature on SysML, there was no complete tutorial to guide people that were new to both the tool and the language. Furthermore, the notation of internal block diagrams supported by the tool was not compliant to the one presented by the text chosen as a reference [55], and this caused a limited

use of these diagrams.
Another issue was the stability of the tool. While the model was growing in size, the tool became slower and more prone to crashes, especially with the increasing number of traceability links between different diagrams. Though this drawback could be associated to the usage of a single model to formalize the whole process, this situation was felt as really frustrating, and led the team to mistrust the tool. As a consequence, many advanced features, such as the collaborative usage, were not experimented. The initial plan was indeed to allow the independent update of the model by different actors in different process phases, but ultimately it was the project leader that took care of the integration of the whole model, according to the input of the other participants.

The final step has been the generation of the documentation. HTML was the preferred format, since with a plain document one would have lost the traceability among artifacts, instead preserved by the hyperlinks. Nevertheless, this choice was criticized by the validation team, on the basis that the format would not have been accepted by the assessors [1]: with a structured document one has a guided direction of reading and understanding, while with HTML one has to choose the navigation path, with the consequent problems of overall uptake. Since the plain document generation capability of the tool was found insufficient, the team had to re-write the documentation by hand, including the SysML diagrams as figures. At this point there were two document sets to maintain, with imaginable versioning problems, and since the SysML models already had their role for the development of the platform, it was decided to keep the textual documentation as the main reference for further changes.
Despite the goal of a complete renewal of the specification and documentation approach was not achieved, the experience did not result in a total failure. The SysML requirement diagrams, used for structuring natural language requirements, have a poor semantics with few connectors (it is not even possible to define requirements with boolean logic relations), and do not give too much added value in themselves with respect to structured paper requirements. However, the possibility to clarify these requirements with other formal diagrams, and to perform mutual tracing, gave a consistent support in requirements disambiguation and

[1] Third-part companies that certify compliance to the CENELEC standard

early discovery of underspecification. The participants agreed that the aid of use case and sequence diagrams as a mean for communication between the requirements manager and the developers simplified the understanding of natural language requirements and increased the level of confidence on the intended behaviour of the system during the implementation. Furthermore, the SysML model worked as a centralized reference for the other activities (e.g., sw/hw development, tests) during the whole project, representing a useful process control tool for the project leader. For these reasons the SysML language survived within the development process of the company, while the TOPCASED tool was soon abandoned in favor of the commercial tool MagicDraw [52].

4.3 The adoption of SysML formal language

As stated before, the need for SysML support came when the systems produced by the company started to radically increase in terms of complexity (to have an idea, when the number of lines of code exceeded 100.000).
SysML have been thus introduced in the initial phases of the development process. The company has defined the current role of SysML in the process, shown schematically in figure 4.14, as follows.

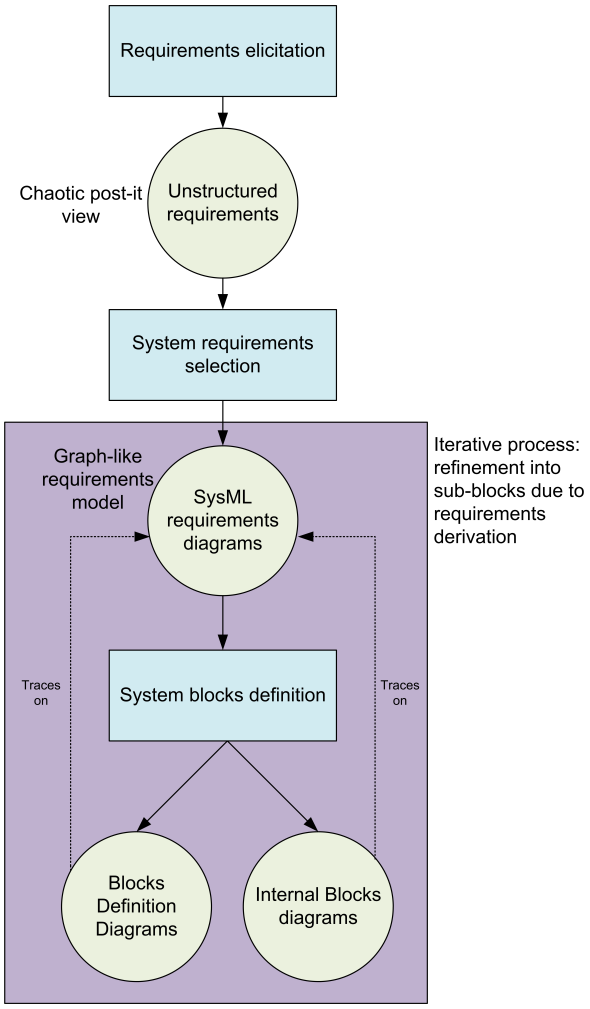


Figure 4.14: SysML role in the requirements definition process

Right after requirement elicitation, requirements appear like unstructured post-it notes in the blackboard of the requirements manager. High-level system requirements are identified among this initial set, and are expressed in the form of SysML requirement diagrams. These diagrams allow specifying hierarchical relationships and dependencies among single requirements, and the chaotic post-it view is replaced by a structured graph-like model.
Then, block diagrams are employed to specify the interfaces of the system modules that are supposed to implement the requirements. An approach based on decomposition is adopted, which allows specializing each module into sub-modules towards the actual implementation. Each module has some high-level require-

ments apportioned to it. If needed, requirements are also refined into lower-level requirements when modules are specialized.
The SysML models are structured into packages residing in a single root model. Each package corresponds to a phase of the V-based development process prescribed by the CENELEC norm for railway safety-critical systems [10]. Therefore, there is a well defined mapping between process phases and the diagrams that have been used in each phase. SysML could be in principle employed also to define the behaviour of the actual implementation and to generate code. However, modelling and simulation at behavioural level is much faster and more flexible with Simulink/Stateflow, and the generated code has higher quality. Therefore, the role of SysML ends at the software architecture level.

4.4 Lesson learned

At the end of the project the general opinion was that using an open-source tool to perform core activities in a company with time-to-market pressure and certification constraints was not a good option for two main reasons:

- Companies prefer products with a limited but stable number of functionalities, while lively maintained opensource tools such as TOPCASED tend to have several experimental features that are progressively tuned by the community according to the users feedback.
- Companies require a direct interface with the tool providers that takes the responsibility if a problem occurs with the tool usage. The choice of Magic Draw was driven by these considerations, and the tool actually confirmed the expectations of a more stable, documented and customer-supported platform.

Nevertheless, the initial goal of passing from text-centric specifications, with diagrams clarifying the text, to diagram-centric ones, with notes accompanying the models, was missed again. Today the company is proficiently employing the Magic Draw platoform on large projects, intensively exploiting collaborative usage features and with a generally good opinion of the tool maturity level, but still all the official specifications required by the CENELEC norms are manually edited

natural language documents. Assessors normally enter at the end of the development process to validate compliance with the standards, and require paper-like documents in order to have a complete picture of the activities performed by the company. While this implies a major effort in terms of production and maintenance of the documentation, it turns out that the investment on SysML pays off in terms of increased confidence on the quality of the specifications.

CHAPTER 5

The Renewed Development Process

This chapter is a critical review of the introduction of advanced software design and verification technologies inside GETS, within a collaboration with the University of Florence. This introduction included the experiences reported in the previous chapters, as well as other PhD experiences, which contributed to the renewal of the development process of General Electric Transportation Systems that is described below.

The transition from a code-based process to a model-based process is not easy. This is particularly true for a company that operates in a safety-critical sector, where the products shall be developed according to international standards, with certified tools and controlled processes.

In this chapter, it will be summarised the experience of a railway signalling manufacturer that decided to adopt general purpose model-based tools, namely Simulink/Stateflow and SysML, for the development of its products. The faced challenges primarily concerned the verification of the software and the integration of the tools within the existing process. Structured development solutions and formal/semi-formal approaches (i.e., semantics restrictions, model-based testing, and abstract interpretation) were adopted to tackle the challenges. The chapter summarise the lessons learnt during this paradigm-shift, with particular focus on

the benefits and drawbacks of automatic code generation.
General Electric Transportation Systems (GETS), signalling division of Florence, is a medium-size railway signalling manufacturer. About ten years ago, given the rising interest of safety-critical industries in formal methods[77], the company decided to start experimenting with formal modelling and verification. To this end, experts were contacted from the university to support the initial experiments, and a strict collaboration have been established in order to develop further the topic of formal methods applied to development process.
Several formal tools were evaluated, but the preference of the developers fell on a semi-formal toolsuite, namely Simulink/Stateflow [82]. The Simulink language uses a block notation for the definition of continuous-time dynamic systems. The Stateflow notation is based on Harel's Statecharts [36] and supports the modelling and animation of event-based discrete-time applications. Simulink block diagrams can be used as a framework to compose Stateflow statecharts, as it is mostly the case for the systems developed by GETS. The main reasons that drove the choice were:

- the large amount of packages available with the toolsuite - packages that could be employed throughout the whole development process;
- the widespread knowledge about the tools found within the company and the corporation.

Initially, the models designed through Simulink/Stateflow were used solely for requirements elicitation. In 2007, the company was attracted by the possibility of using such models also for code generation. One year after, this technology was already part of the development process. However, changing the development paradigm from code-based to model-based required changes also in the verification activities. Model-based testing and abstract interpretation were adopted in the following projects, and strict language restrictions were introduced to constrain the semi-formal semantics of the toolsuite to a formal semantics [26].
The new model-based approach allowed to speed-up the development, and, most of all, gave the possibility to handle more complex systems. As the projects grew in size, new technologies were required to rigorously handle the system requirements and the architecture level of the development. SysML, a unified

modelling language for system development, was selected as the proper technology to address this issue. After three years of experience with SysML, the company has established a formal development approach that integrates SysML and Simulink/Stateflow.

5.1 Challenges

During the story briefly summarized and the experimentation of new formal methods applied to the development of safety-critical systems in GETS, several challenges have been faced that deserve some attention.

Modelling Language Restriction

The code used in safety-critical systems shall conform to specific safety standards, and normally the companies use coding guidelines to avoid usage of improper constructs that might be harmful from the safety point of view. When modelling and auto-coding are adopted, the generated code shall conform to the same standard asked to the hand-crafted code. The adopted code generator, named Simulink Coder, induces a tight relation between the generated code and the modelling language constructs employed. Hence, the identification of a safe subset of the modelling language is required to enable the production of code that is compliant with the guidelines, and that can be succesfully integrated with the existing one. The approach adopted by the company was first to define an internal set of modelling guidelines for Simulink/Stateflow. The guidelines were practical recommendations on the usage of the language constructs. The idea was that the C code generated from models following the guidelines would be compliant with the coding standard of the company. The initial guidelines were based on the analysis of the code that was generated from a model previously designed for requirement elicitation. This preliminary set had the limit of being derived from a specific model, and could lack of generality. Therefore, in the projects that followed, the set was extended with other recommendations borrowed from the automotive domain (i.e., the MAAB guidelines) [24].
In order to ease formal analysis, it was finally decided to complete the modelling style guidelines by restricting the Stateflow language to a semantically unambigu-

ous set. To this end, the studies of Scaife et al. [67], focused on translating a subset of Stateflow into the Lustre formal language, have been used [28]. Models currently developed by the company are therefore independent from the simulation engine, and this choice has actually open the door to formal verification [26].

Generated Code Correctness

Safety-critical norms, such as the CENELEC EN 50128, the European standard for railway software [10], ask for a certified or proven-in-use translator. In absence of such a tool, like in the case of the available code generators for Simulink/Stateflow, a strategy has to be defined to ensure that the code behaviour is fully compliant to the model behaviour, and no additional improper functions are added during the code synthesis phase. The objective is to perform the verification activities at the level of the abstract model, minimizing or automating the operations on the code.
The company has adopted a model-based testing approach called translation validation [17], completed by static analysis by means of abstract interpretation [18]. With translation validation one executes test scenarios based on functional objectives at the model level. Then, he repeats the same tests on the generated code, checking that the outputs of model and corresponding code are consistent. As a final step, in order to ensure runtime error freedom, the Polyspace tool is employed to perform abstract interpretation (see [27] [34] [28] for the details). This technology verifies the correctness of a program on an overapproximation of the range of the program variables.
The certification authorities have considered the overall approach suitable to bypass the tool qualification required by the safety regulations. It should be noticed that the railway norms are not as specific about tool qualification as, for example, the avionic ones [42]. Therefore, companies in the railway sector are required to agree upon possible strategies with the certification authorities.

Multiple Formalisms

Safety-critical systems are normally large, complex platforms with several interacting units and architectural layers. To manage such complexity, their devel-

opment is based on multiple levels of abstraction, and different models with different granularities are required. Indeed, a model that is used for code generation is hardly usable to reason at system design level. Simulink/Stateflow do not support a flexible hierarchical development approach, and system designers need to adopt other modelling languages that can express the higher abstractions inherently required by the process.
In the experience of the company, this issue has been addressed through the adoption of the SysML language [55].

Process integration

Product development is performed by companies by means of processes, which define a framework made of tasks, artifacts and people. Introduction of new technologies in an established process requires adjustments to the process structure, which shall maintain its coherence even if changes are applied. This is particularly true in the case of safety-critical companies, whose products have to be validated according to normative prescriptions. Hence, a sound process shall be defined in order to integrate modelling and code generation within the existing framework.

An enhanced V-based process has been defined, as depicted in Fig. 5.1. The process embeds two verification branches: one for the activities performed on the models, and the other for the tasks concerning source code and system. In the figure, we highlight the parts that strictly concern software development - based on Simulink/Stateflow modelling - and the parts that are related to system development - based on SysML modelling. The two process fragments overlap in the SW Requirements phase and in the SW Model Architecture phase. Indeed, software requirements are expressed in SysML, as well as the software architecture. An equivalent architecture is expressed through Simulink in the form of interacting blocks, which are the functional modules (i.e., the components) of the model. SysML requirements are manually traced to the Simulink model. In the Model Module Design phase, the Simulink blocks are refined into Stateflow statecharts.
It is worth noting that the process is somehow adaptable to both manual coding and to auto-coding. After the SysML modelling activities, one can decide to

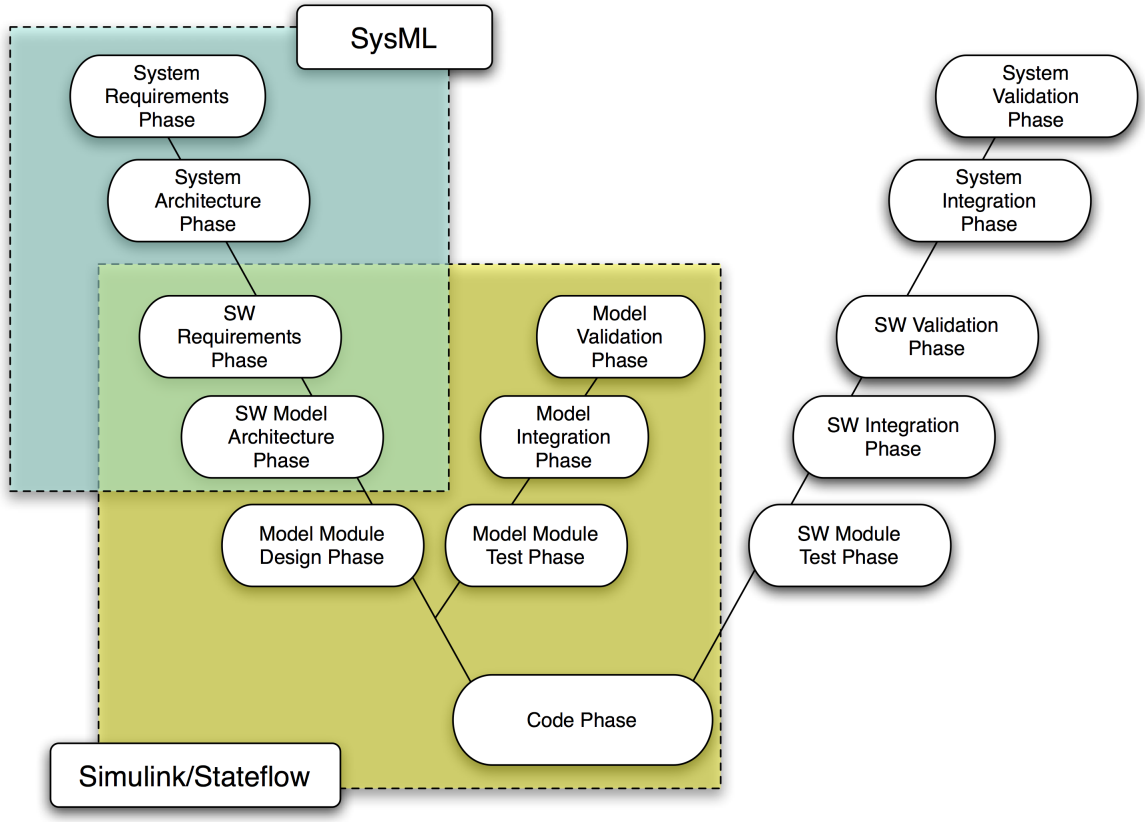


Figure 5.1: The renewed development process

adopt either hand-crafted code or Simulink/Stateflow modelling to develop the application. Indeed, in some applications (e.g., firmware, systems with limited software, platform with strong dependencies from legacy code), the code generation technology is considered not convenient, and hand-crafted code is normally employed.

5.2 Lesson learnt

Facing the challenges of model-based development and formal methods adoption for the overall development process, made the company learn some important lessons, detailed in the following paragraphs.

Abstraction

Models allow working at a higher level of abstraction, and they can be manipulated better than code. The company experienced the actual relevance of this statement in the transition from code-based to model-based testing. The model-based testing approach adopted has allowed defining behavioural test scenarios at component level without disrupting the model structure. This approach would have been impracticable on hand-crafted code. Indeed, with hand-crafted code, it is common to perform tests on single functions, while it is more complex to identify the functions that participate to the behaviour of a software component. With models, one builds the system already in terms of components. Therefore, identification and testing of components comes in a natural way.
The company learnt also that abstraction is a delicate concept that has to be carefully handled. The proper degree of abstraction has to be identified in order for an artifact to be useful. For example, in the initial experience with SysML, requirement diagrams with natural language requirements were adopted throughout the process until the lowest level of model detail. At this point, their content was basically equivalent to the Simulink/Stateflow models. The level of abstraction of such requirements had to be raised, since they appeared to be redundant and any slight modification to the models would have implied a modification to the requirements.

Expressiveness

Graphical models are closer to the natural language requirements. At the same time, they are an unambiguous mean to exchange or pass artifacts among developers. This observation has been enlightened by the main model-based development experience of the company reported in [26], where the project passed from the hands of its first main developer to another developer within one month only and with very limited support.
The previous experience of the company was that, if someone was the father of a piece of software, he would have remained the one and only repository of the knowledge for that software. This is a common problem in many small and average size companies. It negatively affects both the company itself, which has to rely on a single person to actually modify and reuse the software, and the

developer, who normally wish to extend his competencies to go beyond his initial fragment of code.

Cohesion and decoupling

The automatically generated software is composed by modules with higher internal cohesion and better decoupling with respect to manual coding. Interfaces among functionalities are based solely on data, and the control-flow is simplified since there is no cross-call among different modules. Decoupling and well-defined interfaces have helped in easing the outsourcing of the modelling activity, which is a relevant aspect in the development of products that have to tackle time-to-market issues.

Uniformity

The generated code has a repetitive structure, which facilitates the automation of the verification activities. When strict modelling guidelines are defined, one could look at the generated code as if it would be the software always written by the same programmer. Therefore, any code analysis task can be tailored on the artificial programmer's design habits. As a witness for this observation, consider that the abstract interpretation procedure adopted to reveal runtime errors resulted actually profitable on the generated code only, since systematic analysis on hand-crafted code was made harder by its variable structure and programming style.
Uniformity is guaranteed also at the process level with the support of SysML. Employing a unified modelling language - and a single tool - in great part of the development phases eases all those activities that involve the interfaces among the phases. Indeed, in a V-based process, the output artifact of a phase is the input artifact for the following one. The use of SysML has somehow made rigorous this handing over.

Traceability

Software modules are directly traceable with the corresponding blocks of the specification modelled with Simulink/Stateflow. Traceability is a relevant issue in the development of safety-critical systems, since any error has to be traced

back to the process task, or artifact defect, that produced it. The structured development approach introduced, with the support of the Simulink/Stateflow toolsuite, has allowed defining navigable links between the single code statements and the requirements.
At the SysML level, traceability involves the links between requirements diagrams and related SysML diagrams. Traceability links are manually defined through simple drag and drop operations, and traceability matrixes are automatically generated. In a traditional process, traceability matrixes are manually edited, with no tool support and consequent maintainability issues. In the experience of the company, when change requests are issued by a customer, they normally involve system-level requirements. The tool support available with a model-based approach allows tracing the changes from such requirements to the module-level requirements and the corresponding models. Therefore, both the developers and the requirements managers have a complete view of the impact of the change requests. Instead, in a traditional process, one would have to inspect the traceability matrix and check the artifacts that are affected by the change request, an activity that can be rather time consuming and error prone (unless supported by proper automated tools).
Automatic traceability support between SysML and Simulink/Stateflow models is still an open issue, since there is currently no tool that implements such a feature.

Documentation

For safety-critical systems, the official documentation associated to each process phase and artifact is as important as the actual system. The certification of these product is mainly based on the inspection of such a documentation by an external authority. It is therefore important to have a documentation that is formal, expressive and up-to-date with the product status. In the process currently implemented by the company, both SysML and Simulink/Stateflow models are used to provide documentation for the process artifacts. Simulink/Stateflow models with proper comments are used to automatically generate the software documentation. Hence, documentation and software are totally aligned. On the other hand, SysML diagrams are integrated into the manually edited docu-

mentation. Documents can be automatically generated from SysML as HTML pages, but certification authorities normally require to have paper-like documents focused on text, rather than navigable HTML documents with SysML models. The main reason is that the certification authority normally enters at the end of the development process to validate compliance with the standards, and wants to analyse the process as a sequential history - a paper-like document - and not as an interwoven graph of HTML pages.
The integration of the SysML models into the documents poses maintainability issues. Indeed, if the model changes, the change is not automatically reflected by the documentation. However, the one-to-one correspondence between SysML packages and process phases - and associated documents - eases the manual update of the documentation. Furthermore, the traceability links between models in different packages helps the maintenance of the cross dependencies among documents. When a model is changed, the model package clearly identifies the document that has to be modified. Then, one can follow the traceability links to retrieve the other models that are affected by the change. Such models belong to packages with associated documents. Hence, the link among models indirectly create a relationships among documents, and the overall SysML model becomes a sort of navigable index for the process documentation.

Verification cost

The introduction of the new development process has allowed the reduction of the cost of the verification activities, while ensuring greater confidence on the product safety. When passing from traditional code unit testing based on structural coverage objectives, to testing based on functional objectives aided with abstract interpretation, it was possible to reduce the verification cost of about 70%. The new approach was comparable to the previous one in terms compliance to the CENELEC EN 50128 requirements on verification, but resulted much more cost-effective [27].
Though consistent cost improvements have been achieved, manual test definition is still the bottleneck of the process, requiring about 60-70% of the whole unit-level verification cost. Preliminary experiments with formal verification applied at unit-level have shown that this technology might considerably reduce the veri-

fication cost for the majority of the requirements. Indeed, the recent experiments with formal verification by means of Simulink Design Verifier have shown that the verification cost can be further reduced by 50-66% [26].

Control

The structured development has allowed achieving greater control over the components and, finally, to produce software with less bugs as the input to the verification activities. This is witnessed by the number of bugs found by verification, which decreased from 10 to 3 bugs per module when the company introduced a rigorous model-based process.

Complexity

The main drawback encountered in introducing code generation has been the size and overall complexity of the resulting software. Though these aspects were not complicating the verification activities, they posed challenges from the performance point of view.
Real-time constraints for railway signalling systems are not so demanding as are for other kinds of embedded systems, and the typically required response times are in the range of hundreds of milliseconds. However they are reactive systems that, might a failure occur, shall activate failure recovery procedures in a limited amount of time in order to reach the safe state. The reaction time is influenced by the main execution time. In the first experiments with code generation this execution time resulted four times higher compared to the time required by the execution of the corresponding hand-crafted code. In order not to abandon the advantages of auto-coding, in the discussed case an hardware upgrade actually solved the problem. However, during the design of new, more complex systems, this issue has to be taken into account while defining the hardware architecture. The hardware designer shall consider that the code is larger in size, and there is less flexibility in terms of optimizations at source level (we recall that optimizations at compiler level are not recommended for the development of safety-critical systems): when designing the platform, a larger amount of memory has to be planned if one wants to employ automatic code generation.

Knowledge Transfer

Some lessons have been learned also from the knowledge transfer point of view. The research activity has been performed according to the following research management model. On one side there is a research assistant who comes from the university and is fully focused on the technology to be introduced. On the other side there is an internal development team, which puts the research into practice on real projects when the exploratory studies are successful.
The results obtained during this experience would have not been possible through intermittent collaborations only. Moreover, they would have been hardly achieved if just an internal person would have been in charge of the research. In order to separate the research from the time-to-market issues, the independence of the research assistant from the development team has to be preserved. Large companies can profit from dedicated internal research teams, or even entire research divisions. Instead, medium-size companies often have to employ the same personnel for performing research explorations, which are always needed to stay on the market, and for taking care of the day-by-day software development. We argue that the research management model adopted in the presented experience, based on an academic researcher independently operating within a company, can be adapted to other medium-size companies with comparable results.

5.3 Remarks

General Electric Transportation Systems was able to understand the benefits of a model-based process aided with formal methods especially thanks to the initial enthusiasm associated to code generation. Such technology showed its potentials in few months, and its adoption was straightforward. Then, a butterfly-effect in the process occurred that brought to the adoption of other techniques, such as model-based testing, abstract interpretation and system modelling with SysML. Formal verification is not part of the GETS development process yet. We can observe that nevertheless formal verification with model checking is often the subject of the first experiments of companies with something formal, especially in the safety-critical systems domain. In many cases, they did not go much further than these initial experiments, notwithstanding the achieved evidence

of lower verification costs. Indeed, the adoption of formal verification without intermediate steps is not common: the difficulties related to the steep learning curve required by formal methods often tend to discourage industrial practitioners and managers, who need to see the evidence of productivity gains within short time. The reported experience shows that may be more effective to start with less formal tasks (i.e., code generation), and later adopt more formal tasks, such as verification, when the company has matured a full awareness of the actual benefits of "being formal".

Conclusions

The work presented in this dissertation is the result of a research activity aimed to introduce formal methods and code generation techniques in the development process of a railway signalling manufacturer, the General Electric Transportation Systems (GETS) [30], Intelligent Control Systems, division of Florence.
The introduction of new development methodologies in a consolidated process is not a straightforward step: GETS operates in a CENELEC [9] regulated context, and every modification in the development process is likely to be followed by modifications in the verification and validation activities.
At the beginning of the research activity GETS was already experimenting the Model-Based Design (MBD) and code generation by means of Simulink/Stateflow, as result of another PhD research [28]. The issue that the company was facing at that moment was mostly related to the validation of the generated code, and to the integration of the MBD in the existing development process. In order to address this issue, GETS required the support of the Computer Engineering department of the University of Florence.

The code generator provided by the Simulink/Stateflow tool-suite is not certified according to the CENELEC EN 50128 [10]; since the norms ask for a certifid or proven-in-use translator, a strategy has to be defined in order to assess the equivalence between the model and generated code behaviour. The proposed strategy was to fulfill the requirements of the proven-in-use criterion. As a core

part of the process to achieve the proven-in-use property for the translator, it was useful to introduce a technique named translation validation [17]: this technique consists in verifying the functional equivalence between models and generated software by executing the same tests on the model and the code, and afterwards performing further structural analysis to ensure that no additional functionality has been introduced [4]. The Model-Based Testing (MBT) approach that was finally adopted as part of new verification process was an implementation of the translation validation technique. A framework called 2M-TVF, that stands for Matlab Model Translation Validation Framework, fully automated and integrated with the Simulink/Stateflow environment was developed during the experimentation to perform the MBT and the code validation process.
The introduction of abstract interpretation [34] led to a further improvement in terms of development time and verification accuracy.
According to the experience, the cost of formal modeling is slighty higher than manual coding (aboout 30%). This workload increase is partly due to the fact that graphic editing is inherently slower than textual editing, and partly to the training cost required by the technological shift. Nevertherless, the case study shows that this greater effort is payed back by the cost reduction of the code verification activities (about 70% in total, with respect to a manual coding based process) and by the increased confidence on the product safety and quality.

Simulink/Stateflow are suitable for formalizing low-level requirements, but they are less usable for high-level system requirements specification and analysis.
GETS decided to introduce modeling techniques also at a higher level of the development process, in order to substitute the paper-based approach, with natural language documents, inherently ambiguous, completed by informal diagrams, with a more formal one. The OMG SysML language [55] was seen as the solution to substitute the traditional text-centric specifications with a formal notation. The opensource tool TOPCASED was chosen to perform the first experimentation with SysML in a real project, namely the FDT project. SysML model revealed to be a useful process control tool for the project leader. Anyway using an open-source tool to perform core activities in a company with time-to-market pressure was not considered the best choice, mostly for the absence of support and problems related to the tool stability. For these reasons the SysML language

survived within the development process of the company, while the TOPCASED tool was soon abandoned in favor of the commercial tool MagicDraw [52].

The new development process [29] is the result of a long-term effort in introducing formal methods within the company. All the technologies used in the development of the various projects (Metrô Rio project (2009-2011), FDT project (2010-2011)), namely formal modeling, code generation, model-based testing and abstract interpretation, passed a three-stage exam before becoming an internal standard. First, they have been evaluated on an actual product, but offline with respect to the development process, then they have been introduced as part of pilot projects and finally they have been refined and adopted. Formal verification, experimented in the context of Metrô Rio project, is still at the first stage. The possibility of a further reduction of the unit-level verification costs encourages the company to invest in this direction, and activities are currently performed to prepare the subsequent stages. Concerning the formal methods applied to the requirements and architecture specification, the experience of GETS has shown that SysML is an appropriate specification mean in a safety-critical context, but informal natural language is still fundamental to support the evidence that a formal process has been followed.

Bibliography

[1] Abdulla, P. A., Deneux, J., Stalmarck, G., Agren, H., & Akerlund, O. (2004). Designing safe, reliable systems using Scade. In *ISoLA* (pp. 115-129).

[2] American National Standard for Programming Languages (1990). C, ANSI/ISO 9899

[3] Laprie, J.C., Avizienis, A., & Kopetz, H. (1992). Dependability: Basic Concepts and Terminology. Springer-Verlag New York

[4] Baresel, A. and Conrad, M. and Sadeghipour, S. & Wegener, J. (2003). The interplay between model coverage and code coverage. In *Proceedings 11th European conference on Software Testing, Analysis and Review (EuroSTAR'03)*. Amsterdam, the Netherlands.

[5] Boca, P.P., Bowen, J.P., & Siddiqui, J. I. (2009). Formal Methods: State of the Art and New Directions. London, UK: Springer.

[6] Bacherini, S., Fantechi, A., Tempestini, M. & Zingoni, N. (2006). A Story about Formal Methods Adoption by a Railway Signaling Manufacturer. In *LNCS 4085: Proceedings of the 14th International Symposium on Formal Methods* (FM 2006) (pp. 179-189), Hamilton, Canada. Berlin, Germany: Springer.

[7] Bochot, T., Virelizier, P., Waeselynck, H., & Wiels, V (2009). Model checking flight control systems: The Airbus experience. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), Companion Volume* (pp. 18-27), Vancouver, Canada. Washington D.C.: IEEE Computer Society.

[8] Blanchet, B, Cousot, P., Cousot, R., Feret, R., Mauborgne, R., Min, A., Monniaux, D., & Rival, X. (2003). A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (pp. 196-207), San Diego, CA. New York, NY: ACM Press.

[9] CENELEC - European Committee for Electrotechnical Standardisation (1994). EN 50126, Railway Applications - The specification and demonstration of dependability, reliability, availability, maintainability and safety.

[10] CENELEC - European Committee for Electrotechnical Standardisation (1996). EN 50128, Railway Applications - Software for railway control and protection systems.

[11] CENELEC - European Committee for Electrotechnical Standardisation (1994). EN 50129, Railway Applications - Safety related electronic railway control and protection systems.

[12] CENELEC - European Committee for Electrotechnical Standardisation (1996). EN 50159, Railway Applications - Communication, signaling and processing systems, Part 1: Safety-Related Communication in Closed Transmission Systems.

[13] Chang, C.L., Lee, R.C., & LeeF, R.C.T. (1997). Symbolic Logic and Mechanical Theorem Proving. London, UK: Academic Press, Inc.

[14] Clarke, E.M., Grumberg, O., & Long, D.E. (1994). Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5): 1512-1542.

[15] Clarke, E. M., Wing, J. M., Alur, R., Cleaveland, R., Dill, D., Emerson, D., Garland, S., & al. (1996). Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4): 626 - 643.

[16] Clarke, E. M., Biere, A., Raimi, R., & Zhu, Y. (2001). Bounded model checking using satisfiability solving, *Formal Methods in System Design* (pp 7-34).

[17] Conrad, M. (2009). Testing-based Translation Validation of Generated Code in the Context of IEC 61508. *Formal Methods in System Design*, 35(3): 389-401.

[18] Cousot, P., & Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction for approximation of fixpoints. In *Proceedings of POPL* (pp. 238-252).

[19] Dormoy, F.X. (2008). Scade 6: a model based solution for safety critical software development. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS08)* (pp. 19), Toulouse, France.

[20] ETAS - Ascet Software Products. Retrieved December 20th, 2013, from http://www.etas.com/.

[21] dSPACE GmbH TargetLink Product Management (2008). Modeling Guidelines for MATLAB/Simulink/Stateflow and TargetLink. Retrieved December 28th, 2013, from https://www.e-guidelines.de.

[22] Farkas, T., Hein, C., & Ritter, T. (2006). Automatic Evaluation of Modelling Rules and Design. *Second Workshop: From code centric to model centric software engineering: Practices, Implications and ROI*. Bilbao, Spagna.

[23] Faivre, A., & Benoit, P. (1999). Safety critical software of meteor developed with the B formal method and vital coded processor. Proceedings of the World Congress on Railway Research (WCRR99), (pp. 84-89).

[24] Ferrari, A., Fantechi, A., Tempestini, M., & Zingoni, N. (2009). Modeling Guidelines for Code Generation in the Railway Signaling Context. In *Proceedings of 1st NASA Formal Methods Symphosium (NFM)* (pp 166-170). Moffet Field, CA, USA.

[25] Ferrari, A., Fantechi, A., Papini, M., & Grasso, D. (2010). An industrial application of formal model based development: the Metrô Rio ATP case. In *Proceedings of the 2nd International Workshop on Software Engineering for Resilient Systems* (SERENE 2010) (pp. 71-76). London, UK.

[26] Ferrari, A., Grasso, D., Magnani, G., Fantechi, A., & Tempestini, M. (2010). The Metrô Rio ATP case study. In *LNCS 6371: 15th International Workshop on Formal Methods for Industrial Critical Systems (FMICS2010)* (pp. 1-16). Antwerp, Belgium. Berlin, Germany: Springer.

[27] Ferrari, A., Magnani, G., Grasso, D., Fantechi, A., & Tempestini, M. (2011). Adoption of Model-based Testing and Abstract Interpretation by a Railway Signalling Manufacturer. *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)* 2(2): 42-61.

[28] Ferrari, A. (2011). Adoption of Code Generation by a Railway Signalling Manufacturer. PhD thesis. University of Florence.

[29] Ferrari, A., Fantechi, A., Gnesi, S., & Magnani, G. (2013). Model-Based Development and Formal Methods in the Railway Industry. *Software, IEEE*, 30(2): 28-34.

[30] General Electric Transportation Systems. Retrieved December 20th, 2010, from http://www.getransportation.com/.

[31] GNU Project. GNU Compiler Collection (GCC), Version 3.4.3. Retrieved December 17th, 2013, from http://gcc.gnu.org/gcc-3.4/.

[32] Gordon, M.J.C., & Melham, T.F. (1993). Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge, UK: Cambridge University Press.

[33] Grasso, D., Fantechi, A., Ferrari, A., Becheri, C., & Bacherini, S. (2010). Model Based Testing and Abstract Interpretation in the Railway Signaling Context. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation* (pp. 103-106). Paris, France. Washington D.C.: IEEE Computer Society.

[34] Grasso, D. (2014). Testing and verification methods for Safety Critical systems. PhD thesis. University of Florence.

[35] Guiho, G., & Hennebert, C. (1990). Sacem software validation. In *Proceedings 12th int. conf. on Software engineering* (pp. 186-191). ICSE '90, IEEE Computer Society Press, Los Alamitos, CA, USA.

[36] Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3): 231274.

[37] Holzmann, G.J. (2004). The SPIN Model Checker: Primer and Reference Manual. Lucent Tehcnologies Inc., Bell Laboratories.

[38] Huber, F., Schaatz, B., Schmidt, A., & Spies, K (1996). Autofocus - a tool for distributed systems specification. In *LNCS 1135: Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT996)* (pp. 467-470). Berlin, Germany: Springer.

[39] International Council on Systems Engineering. Retrieved December 28th, 2013, http://www.incose.org/.

[40] ISO/IEC 8652 (1995). Information technology - Programming languages - Ada.

[41] Jones, C. B. (1990). Systematic Software Development using VDM, 2nd ed. Englewood Cliffs, NJ: Prentice Hall.

[42] Kornecki, A. J., & Zalewski, J. (2009). Certification of software for real-time safety-critical systems: state of the art. ISSE, 5(2): 149-161.

[43] Leuschel, M., Falampin, J., Fritz, F., & Plagge, D. (2009). Automated property verification for large scale B models. In *LNCS 5850: FM 2009, Formal Methods* (pp. 708-723). Springer.

[44] Mathworks Automotive Advisory Board (MAAB) (2007). Control Algorithm Modeling Guidelines Using Matlab, Simulink and Stateflow, Version 2.0. Retrieved December 17th, 2012, from http://www.mathworks.com/industries/auto/maab.html

[45] Meenakshi, B., Bhatnagar, A., & Roy, S. (2006). Tool for translating Simulink models into input language of a model checker. In *LNCS 4260: Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM 2006)* (pp. 606620). Berlin, Germany: Springer.

[46] Milius, S., Steinke, U.: *Model-driven Software Development: Needs and Experiences in Rail Automation*. Siemens AG, FMICS2008

[47] Miller, S. P., Whalen, M. W., & Cofer, D. D. (2010). Software Model Checking Takes Off. *Communications of the ACM* 53(2): 58-64.

[48] Mohagheghi, P., Dehlen, V. (2010). Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In *LNCS 5095: Model Based Testing of Reactive Systems (pp. 432-443)*. Berlin, Germany: Springer.

[49] Monin, J. F. (2003). Understanding Formal Methods. London, GB: Springer.

[50] Model Engineering Solutions (MES). Model Examiner. Retrieved December 20th, 2010, from http://www.model-engineers.com/en/our-products/model-examiner.html.

[51] Nielsen, M., Havelund, K., Wagner, K., & George, C. (1989). The RAISE language, method and tools. *Formal Aspects of Computing* 1: 85-114.

[52] No Magic, Inc. Magic Draw. Retrieved December 27th, 2013, from http://www.nomagic.com/products/magicdraw.html.

[53] Object Management Group. Retrieved December 28th, 2013, from http://www.omg.org/.

[54] Object Management Group. UML2 Specification. Retrieved December 20th, 2010, from http://www.omg.org/spec/UML/2.0/

[55] Object Management Group (2010). OMG Systems Modeling Language (OMG SysML), Version 1.2. Retrieved September 22nd, 2012, from http://www.omg.org/spec/SysML/1.2

[56] OSLO Project - Open Source Library for OCL. Retrieved Semptember 25th, 2011, from http://oslo-project.berlios.de

[57] Osherove, R. (2009). The art of unit testing. Manning.

[58] Owre, S., Rushby, J. M., & N. Shankar (1992). PVS: A prototype verification system. In *LNCS 607: Automated DeductionCADE-11, Proceedings of the 11th International Conference on Automated Deduction* (pp. 748-752), Saratoga Springs, NY, USA. Berlin, Germany: Springer.

[59] Papadopoulos, Y., McDermid, J.A. (1999) The Potential for a Generic Approach to Certification of Safety-Critical Systems in the Transportation Sector. In *Journal of Reliability Engineering and System Safety* (pp. 47-66), Vol. 63, No. 1.

[60] Pnueli, A., Siegel, M. & Singerman, E. (1998). Translation Validation. In *LNCS 1384: Proceedings 4th international conference on Tools and Algorithms for the Construction and Analysis of System* (pp 151-166). Lisbon, Portugal. Berlin, Germany: Springer.

[61] Radio Technical Commission for Aeronautics (1992). DO-178B - Software Consideration in Airborne Systems and Equipment Certification.

[62] Rete Ferroviaria Italiana (RFI) (2008). Specifica dei Requisiti di Sistema SCMT, Vol. 3, Sottosistema di Bordo Appendice A - Requisiti Funzionali

[63] Rete Ferroviaria Italiana (RFI) (2008). Specifica dei Requisiti di Sistema SCMT, Vol. 3, Sottosistema di Bordo Appendice A - Allegato 11 - Presenza PdM.

[64] Rete Ferroviaria Italiana (RFI) (2008). Specifica Requisiti di Sistema SCMT, Vol. 2, Sottosistema di terra.

[65] Rete Ferroviaria Italiana (RFI) (2008). Specifica Requisiti di Sistema SCMT, Vol. 3, Sottosistema di bordo.

[66] Ricardo, Inc., MINT: *http://www.ricardo.com/mint*. 2008

[67] Scaife, N., et al. (2004). Defining and translating a safe subset of Simulink/Stateflow into Lustre. In *Proceedings of the 4th ACM International Conference on Embedded Software* (pp. 259-268), Pisa, Italy.

[68] Scilab - The Free Platform for Numerical Computation. Retrieved December 20th, 2013, from http://www.scilab.org/.

[69] Schneider, S. (2001). The B-method: an introduction. Basingstoke, UK: Palgrave Macmillan.

[70] Schreiber, F., Morzenti, A. (1999). Problematiche di certificazione del software safety-critical. Parte I - Generalità, elicitazione e specifica dei requisiti. In *Ingegneria Ferroviaria*, Anno LIV, No. 10 (pp. 709-746).

[71] Selic, B. (2003). The Pragmatics of Model-driven Development. *IEEE Software* , 20(5): 19-25.

[72] Sheeran, M., Singh, S., & Stlmarck, G. (2000). Checking safety properties using induction and a sat-solver. In *LNCS 1954: Formal Methods in Computer-Aided Design* (pp. 127-144). Springer.

[73] Spivey, J.M. (1988). Understanding Z: a specification language and its formal semantics - (Cambridge Tracts in Theoretical Computer Science Vol. 3). Cambridge, UK: Cambridge University Press.

[74] Sturmer, I., & Travkin, D. (2007). Automated Transformation of MATLAB Simulink and Stateflow Models. In *Proceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-time Systems (OMER2007)* (pp. 57-62), Paderborn, Germany.

[75] Sturmer, I., Kreuz, I., Schafer, W., Schurr, A. (2007). The MATE Approach: Enhanced Simulink and Stateflow Model Transformation. In *Proceedings of MathWorks Automotive Conference (MAC 2007)*, Dearborn, MI, USA.

[76] Vouk, M. A. (1990). Back-to-back Testing. Information and Software Technology, 32(1): 34-45.

[77] Woodcock, J., Larsen, P.G., Bicarregui, J., & Fitzgerald, J. (2009). Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4): 1-36.

[78] The MathWorks. Airbus User Story. Retrieved December 27th, 2013, from http://www.mathworks.it/company/user_stories/Airbus-Develops-Fuel-Management-System-for-the-A380-Using-Model-Based-Design.html

[79] The MathWorks. Alstom User Story. Retrieved December 27th, 2013, from http://www.mathworks.it/company/user_stories/Alstom-Generates-Production-Code-for-Safety-Critical-Power-Converter-Control-Systems.html

[80] The MathWorks. Real Time Workshop Embedded Coder. Retrieved December 20th, 2010, from http://www.mathworks.com/products/rtwembedded/.

[81] The MathWorks. Stateflow Coder. Retrieved December 27th, 2013, from http://www.mathworks.com/products/sfcoder/.

[82] The MathWorks. Simulink. Retrieved December 27th, 2013, from http://www.mathworks.com/products/simulink/.

[83] The MathWorks. Simulink Design Verifier. Retrieved December 27th, 2013, from http://www.mathworks.com/products/sldesignverifier/

[84] The MathWorks. NASA User Story. Retrieved December 27th, 2013, from http://www.mathworks.it/company/user_stories/NASA-Uses-Stateflow-and-Simulink-Coder-to-Generate-Fault-Protection-Code-for-Deep-Space-1.html

[85] TOPCASED. Retrieved December 27th, 2013, http://www.topcased.org/.

[86] Tretmans, J., Wijbrans, K., Chaudron, M.R.W. (2001). Software Engineering with Formal Methods: The Development of a Storm Surge Barrier Control System Revisiting Seven Myths of Formal Methods. *Formal Methods in System Design*, 19(2): 195-215.